The Replica Consistency Problem in Data Grids

Gianni Pucciani

A thesis submitted for the degree of Doctor of Philosophy in Information Engineering

Department of Information Engineering
University of Pisa

February 2008

CERN-THESIS-2008-049
01/02/2008

Abstract

Fast and reliable data access is a crucial aspect in distributed computing and is often achieved using data replication techniques. In Grid architectures, data are replicated in many nodes of the Grid, and users usually access the "best" replica in terms of availability and network latency. When replicas are modifiable, a change made to one replica will break the consistency with the other replicas that, at that point, become stale. Replica synchronisation protocols exist and are applied in several distributed architectures, for example in distributed databases. Grid middleware solutions provide well established support for replicating data. Nevertheless, replicas are still considered read-only, and no support is provided to the user for updating a replica while maintaining the consistency with the other replicas.

In this thesis, done in collaboration with the Italian National Institute of Nuclear Physics (INFN) and the European Organisation for Nuclear Research (CERN), we study the replica consistency problem in Grid computing and propose a service, called CONStanza, that is able to synchronise both files and heterogeneous (different vendors) databases in a Grid environment. We analyse and implement a specific use case that arises in high energy Physics, where conditions databases are replicated using databases of different makes. We provide detailed performance results, and show how CONStanza can be used together with Oracle Streams to provide multi-tier replication of conditions databases using Oracle and MySQL databases.

Ai miei genitori, e a Karolin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information management is becoming increasingly critical in today's IT infrastructures. More and more data need to be stored, and ever more complex analyses are performed on these data. In addition, data access must be efficient and without interruptions.

In research environments, as well as in the industry, Grids are used to handle new computing challenges. In a Grid, many different institutions share computing and data storage resources, and collaborate to perform data and CPU intensive tasks. For example, at CERN (European Organization for Nuclear Research), a Grid infrastructure is used to store and analyse huge amounts of data coming from a particle accelerator in order to discover new elementary particles. Data are generated by particle detectors, and shipped to world-wide distributed data analysis centres where the actual analyses are performed.

In Grids, as well as in other distributed computing platforms, data are replicated, i.e., copied at different locations, in order to improve the performance and the reliability of the applications. However, the replication mechanisms developed so far in Grid computing can be considered incomplete, in that they do not provide support for *replica consistency*. This means that when a copy of a data item is modified by a user or an application, the modification is not propagated to the other copies (replicas) of the same data item, creating an inconsistency among the replicas.

In this thesis, done in collaboration with the Italian Institute for Nuclear Physics (INFN) and CERN, we investigate the possibility to fill this gap by providing a Replica Consistency Service for Grids. Replica consistency is an application specific topic; it has already been dealt with in distributed databases, and in many other distributed architectures, using different approaches, but no solutions are currently

implemented in Grid computing. This is why, in the first part of this thesis, we concentrate on existing solutions from other fields, and evaluate the possibility of their application to a Grid environment. In the second part instead, we present the design and the implementation of a novel Replica Consistency Service for Grids.

The two main contributions of this thesis are the theoretical study of the replica consistency problem in Grid computing, where only few efforts have been done so far, and the development of a Grid software to solve this problem. The Replica Consistency Service we developed has been successfully tested for a specific use case that arises in High Energy Physics (HEP), that is the synchronisation of heterogeneous databases. A performance analysis has also been done in order to evaluate the performance of the software and drive future development.

1.1 Structure and contents overview

This thesis is organised in two main parts. In the first one, that comprises Chapters 2 through 5, we introduce the application domain and the state of the art. We review the main concepts about replication and consistency, and present some practical applications. We also discuss the feasibility and the main issues in developing a consistency mechanism for Grids. In the second part, from Chapter 6 to Chapter 8 we present the main deliverable of this work, CONStanza, a Replica Consistency Service for Grids. The design and the implementation of this service are explained. A detailed performance analysis is presented to help driving future developments. A functional test for a specific use case is presented, where CONStanza is used together with Oracle Streams to consistently distribute data in a tiered architecture.

The chapters are organised as follows. In Chapter 2, the theory of concurrency control in centralised and distributed databases is reviewed. Database theory is a solid and well understood discipline and it presents many analogies with the problem we are facing in Grid computing. In Chapter 3, we introduce the main mechanisms used in wide area data stores to enforce replica consistency. We focus on optimistic replication, and we characterise different options based on the consistency level that they offer. In Chapter 4, the main application field that drove this work, particle Physics (or HEP) and the experiments performed at CERN, are presented. We focus on the ATLAS experiment, and we present the techniques used to distribute data from CERN to world-wide distributed partner institutions. In Chapter 5, Grid computing is explained, highlighting the peculiarities that make it a novel architecture for distributed and collaborative computing. The software used to manage a Grid is presented, with a focus on data replication services. In this chapter we show the ex-

isting gap in replication services for what concern data synchronisation. In Chapter 6, the main deliverable of these Ph.D. studies is presented: a Replica Synchronisation Service for Grids called CONStanza. We present its development from requirements collection to design and implementation. In Chapter 7, performance and scalability of CONStanza are analysed using a two-factor full factorial experimental design. This analysis allowed us to discover possible bottlenecks of the architecture and to drive future developments. In Chapter 8, we present a functional test that shows how CONStanza can be integrated with the Oracle Streams technology to provide multi-tier data replication with heterogeneous databases. Finally, in Chapter 9, we conclude summarising the main results and describing some interesting work that could follow from this thesis.

Part I

Foundations

Chapter 2

Consistency in centralised and distributed databases

Database systems play a fundamental role in every IT infrastructure; information is critical and its correctness and reliability must be guaranteed without interruptions. Software applications, even when they are carefully designed, well implemented and tested, can suffer from inadequate data management. Concurrent access and failures are the main sources of problems. Nowadays, database technologies are robust enough to be used in critical activities like banking and airlines as well as in advanced research institutes. Databases allow these activities to perform well even when thousands of users access data at the same time, or when hardware failures or disasters cause the unavailability of storage and processing devices. Despite all this, databases are robust enough not to cause service interruptions.

The database theory can be used as a starting point of our problem analysis; it is a solid and well understood discipline, and it presents many analogies with the problem we are faced with, the consistency of replicated data in Grid computing. However, as we will see in Section 5.1, Grid environments present some more issues that databases systems often do not have.

In this chapter we start reviewing the basic concepts of the database theory in Section 2.1. Then, in Section 2.2, we recall the main techniques used in centralised databases to enforce consistency and reliability. Finally, in Section 2.3, we see how these techniques can be extended to guarantee consistency and reliability in distributed databases.

2.1 Concepts and definitions

Users access data stored in a database (DB) by means of *database operations* like $read(x)$ (reading the data item x) and $write(x, val)$ (writing the value val into the data item x) managed by a software application, the Database Management System (DBMS)[1]. In this section and in the rest of the thesis, we consider relational databases[2], that is, databases conform to the relational model [52] where a collection of data items is organised as a set of formally-described tables from which data can be queried using the Structured Query Language (SQL).

Database operations are executed atomically, meaning that they can be considered as units of execution, not decomposable into smaller operations. A set of logically correlated database operations can be grouped to form a *transaction*. Transaction operations, like $start$, *commit* and *abort*, are also atomic and they are used to manage transactions. A transaction can be seen also as an execution of one or more programs that include database and transaction operations.

At this point, before providing further definitions, it is important to recall the basic ACID properties of a transaction [72]: atomicity, consistency, isolation and durability. No database that fails to provide the ACID properties can be considered reliable.

Atomicity Atomicity states that a transaction is treated as a unit of operation. It is also called the "all-or-nothing property" and it involves the problem of state recovery in case of failures happened during the transaction's operations.

Consistency The consistency property says that a transaction should move the database from one consistent status to another, where the consistent status is defined in terms of integrity constraints.

Isolation The isolation property requires that each transaction will not reveal its results to other concurrent transactions before its commitment. This is a necessary condition to avoid that a transaction that is going to abort could provide data read by other concurrent transactions.

Durability Durability ensures that the results of a transaction, after its commit, are permanent and cannot be erased from the database.

After seeing the ACID properties, let us continue to recall some basic concepts of the database theory.

[1] Oracle and MySQL are two examples of DBMS that we will discuss later in this thesis.

[2] Other database models exist, like *object-oriented* and *object-relational*.

2.1.1 Recoverability

A database should perform its operations in such a way that, even in case of a transaction abort, it is able to undo the effects of the aborted transaction allowing active transactions to continue their execution. A database should always behave as if it contains all the results of committed transactions and none of the results of uncommitted ones. Recoverable executions, cascadelessness and strict executions are important concepts in defining the recoverability of a database.

Recoverable executions A recoverable execution is one where a transaction T cannot commit until all transactions that wrote values read by T are themselves committed. We can also say that an execution is recoverable when, for every T that commits, T's commit follows the commit of every transactions from which T reads. Usually delaying the processing of certain commit operations is one way the DBMS can ensure that executions are recoverable. This includes delaying not just write operations on the database, but also output operations that may display intermediate or uncommitted results. In fact, a user might inadvertently use such information as input to further transactions that would fail or be incorrect if the original transaction aborts.

Cascadelessness A DBMS avoids cascading aborts[3] (or is cascadeless) if it ensures that every transaction reads only those values written by committed transactions. Thus, only committed transactions can affect other transactions.

Strict executions Strict executions are those in which both reads and writes for a data item x are delayed until all previous transactions that wrote into x are committed or aborted. It has been proven that *strict executions avoid cascading aborts and are recoverable* [79].

2.1.2 Serializability

Concurrency control problems are entirely due to the concurrent execution of some transactions that operate on the same data items. It is a different problem from Recoverability, that we saw in Section 2.1.1, and can happen also when there are no transaction aborts. Two of the most common problems of concurrent access are *lost updates* and *inconsistent retrieval*. Lost updates happen when two transactions read the same data item and update it one after the other: in this case the second transaction

[3] A cascading abort happens when a single transaction abort leads to a series of transaction rollback of those transactions that used values written by the aborted transaction.

overwrites the value written by the first one, whose update is lost. An inconsistent retrieval happens when a transaction reads a data item which, in the meantime, has been modified by another transaction that has not yet committed. Thus, the first transaction reads a stale data item, and can perform write operations based on this stale information.

Since transactions that execute serially (*serial executions*) do not interfere with each other, they are "correct" from the point of view of the concurrency control, and do not leave space to problems such as lost updates or inconsistent retrieval. Hence, the goal of a concurrency control mechanism would be that of arranging transactions so that they execute serially, but in this case we would loose the benefit of concurrency.

On the other hand, if an execution has the same effects and produce the same output of a serial execution, it is "correct" too. In this case the execution is called *serializable* (SR). Thus, serializability is seen as the correctness property for concurrency control in DBs.

2.1.2.1 Consistency preservation

When dealing with database consistency, it is important to make a distinction between *internal* and *external* consistency. Internal consistency concerns a single database, while external consistency is used to express the consistency among replicated databases. In what follows we deal with internal consistency; external consistency is introduced in Section 2.3, where we review the consistency of distributed databases.

One can define a "consistency predicate" $p()$ about a particular state of the DB in terms of the integrity constraints defined on the database. The DB is *consistent* if and only if the predicate $p(DB)$ is true. So, we can say that each transaction is correct if it *preserves the DB consistency*. Thus, if each transaction preserves the DB consistency, so does any serial execution, and hence also any serializable execution. To resume, *serializability ensures DB consistency*.

It might happen that serializability is not appropriate for some concurrency control problem, either because the concept of transaction does not apply or because, due to the purpose of the application, a certain degree of inconsistency can be tolerated by the system to have a gain in terms of the overall performance. We will see some of these examples in Section 2.3.3, when we talk about lazy replicated databases.

2.2 Techniques used in centralised databases

In this section we illustrate some approaches to build serializable schedulers[4]. We do not provide details but just the main idea that is behind each technique to help the discussion in later chapters.

A *history* (H) is the description of a concurrent execution of a set of transactions and it is defined as a partial order that must preserve the partial order of any pair of operations that are ordered within each transaction [79]. It is also required that a history specifies the order of all *conflicting operations*. Two operations are conflicting if they both operate on the same data item and at least one of them is a write. A history that preserves the order of any pair of operations of its transactions and that specifies the order of all conflicting operations is called a *complete history*. The *committed projection* of a history, instead, is a complete history defined over the set of committed transactions in that history.

From what has been said in the previous section, to see whether an execution is correct, we need to verify its equivalence to a serializable one. In order to do this we need to state the concept of *equivalence of histories*: two histories are equivalent when they are defined over the same set of transactions and they order conflicting operations of non-aborted transactions in the same way. It follows that a history is serializable if its committed projection is equivalent to a serial history, that is a history where transactions are executed serially, one after the other.

The serializability theory makes use of so called *serialisation graphs* (SG). Serialisation graphs are graphs whose nodes are the transactions and whose arcs are the order relations between two transactions. In a serialisation graph, a transaction T_i *happens before* another transaction T_j when one of T_i's operations precedes and conflicts with one of T_j's operations. Using the serialisation graphs, the serializability theorem [79] says that "*a history H is serializable iff its serialisation graph is acyclic*".

In the presence of failures, in order to ensure the correctness of some executions it is not sufficient to verify that they are serializable, they must also be *recoverable*.

It is worth recalling three other important results:

- *A history is recoverable (RC) if each transaction commits after the commitment of all transactions from which it reads.*
- *A history avoids cascading aborts (ACA) when its transactions may read only*

[4]The scheduler is the component of a DBMS that is in charge of arranging the execution of transaction operations so that the database is consistent and recoverable.

those values that are written by committed transactions or by itself.

- *A history is strict (ST) when no data item may be read or overwritten until the transaction that previously wrote into it commits or aborts.*

A history can be RC but not ACA, or ACA but not ST. As regards these properties we can say that **ST** $\subset$ **ACA** $\subset$ **RC**. The SR property intersects all the previous sets, as shown in Figure 2.1.

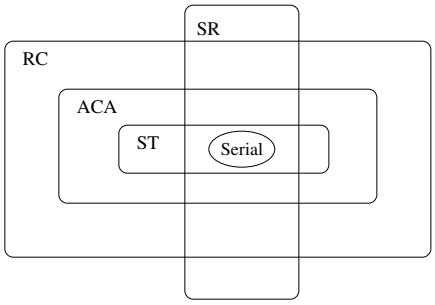

Figure 2.1: Relationships among history properties

Every time we want to build the serialisation graph to verify that it is acyclic (and hence the history is serializable) we have to specify the *compatibility matrix* among each of the possible database operations to find out when conflicts occur.

We must also note that the previous definition of equivalence is known as *conflict equivalence*. There is another way to state the equivalence of two histories, known as *view equivalence*. Since most of the concurrency control algorithms are *conflict based* we do not discuss further the view equivalence.

2.2.1 Locking schedulers

Locking schedulers are the most popular in commercial products. They are generally based on the *two-phase locking* (2PL) algorithm or one of its variations, that we review in the next sections. Two locks conflict if they are issued by different transactions, on the same data item and they are of conflicting type (e.g, at least one is a write lock). It is important to consider the distinction between aggressive schedulers (tend to schedule operations as soon as they arrive, optimistically) and conservative schedulers (tend to delay operations, pessimistically, the extreme case of this being a serial

scheduler). Knowing in advance the readset[5] or the writeset[6] of each transaction may be very useful in building aggressive schedulers. However, in many applications, readsets and writesets are not know in advance. The next sections review the main concepts behind 2PL algorithms without going to much into the details of their specifications.

2.2.1.1 Basic 2PL

The basic version of the 2PL algorithm uses two phases: in the *growing phase* the transaction obtains the locks it needs and in the *shrinking phase* it releases the locks. In this version the only limitation for a transaction is that it may not get any more locks when it has already released one. This is not sufficient to avoid *deadlocks* since locks from different transactions may interleave. One way to avoid deadlock is using *timeouts*, although they can fail assuming the presence of a deadlock or become too expensive when they are set to high values, so that their tuning is a fundamental operation. Another approach for contrasting deadlocks is to detect them by using the *Wait-For Graph* (WFG). In this graph there is a line from T_1 to T_2 if T_1 is waiting for T_2 to release some locks. When the WFG has a cycle, it means that the system has a deadlock. The period of the WFG check must also be correctly tuned. Another important issue is the selection of the *victim* when a deadlock has been discovered; one of the transactions involved in the deadlock must be aborted. Different policies exist for choosing the victim but they do not add anything relevant to the scope of this thesis.

2.2.1.2 Variations of 2PL

A variation of the basic 2PL , called *conservative 2PL*, imposes that a transaction acquires all locks it needs before executing any operation, that is predeclaring its readset and writeset. This way deadlocks are avoided.

Almost all implementations of the 2PL use a variant called *strict 2PL*. This technique does not avoid deadlocks but it ensures *strict* executions (that is RC and ACA, see Section 2.2). It differs from the basic 2PL in that it requires the scheduler to release all of a transaction's locks together, when the transaction terminates.

In some cases effective improvements in the concurrent execution can be obtained if we define some new special operations (like increment(x) or decrement(x), that

[5]The readset of a transaction is the set of data iitems read by its operations.

[6]The writeset of a transaction is the set of data items written by its operations.

specialise the more general form write(x)) with new types of locks and we build a new compatibility matrix.

In designing a concurrency control mechanism the *granularity* of the object being accessed and the locking technique can lead to choose different types of schedulers. A trade-off between concurrency and lock overhead exists as regards the optimal granularity and it depends highly on the application. There is also the possibility to use *multigranularity locking*. This mechanism usually requires the application to be aware of the data that the transaction wants to access and thereby be able to choose the most appropriate lock granularity.

Using 2PL or some of its variations, sometimes breaking long transactions that access several data items into smaller ones can provide some advantages and simplify the design.

2.2.1.3 Tree locking (TL)

This mechanism differs from the 2PL (and its variations) in that it assumes that data are organised in a hierarchical structure and locks can be acquired also after another lock on a different data item has been released. The key rule is that a transaction can lock a data item only if it holds the lock on its parent. It can be proved that tree locking produces serializable executions and avoid deadlocks. TL only makes sense in those applications where the access pattern of a transaction is known, so that, when a lock on a data item is released, we are sure that no children of that data item will be subsequently accessed. In this case we are able to provide better performance (meaning more concurrency) than 2PL since locks can be released earlier. However, the benefits is only realised if transactions normally access data in root-to-leaf order as organised in the data tree.

As regards the recoverability we need to impose further rules on the simple TL. For example, to avoid cascading aborts each lock on a data item must be held until the transaction commits, and this has a serious impact on performance. Variations of TL exist to reduce the lock contention.

2.2.2 Non-locking schedulers

We will see shortly some of the main techniques that do not use locking. They are mostly theoretical since they are not used by commercial DBMS but they provide useful ideas for the purpose of this thesis.

2.2.2.1 Timestamp ordering (TO)

It assigns a unique timestamp to each incoming transaction. Every operation takes the timestamp of the transaction it belongs to. For every data item the value of the maximum timestamp of both read and write operations is recorded. An operation on a data item is executed only if its timestamp is greater than the maximum timestamp of a conflicting operation on the same data. If the timestamp is less, the operation is rejected causing the transaction to abort. Then, the aborted transaction, when resubmitted, will have a greater timestamp avoiding further aborts. Executions of operations must be carried out using a queue and acknowledgements[7] to ensure the correct order. Thus, the scheduler produces serializable executions (refer to [79] for a formal proof).

However, TO does not ensure recoverability, since it can happen that an operation reads a value from a not yet committed transaction. A strict version of TO exists that provides strict executions with a slight change in the algorithm. The size of the data structure used for storing timestamp values is proportional to the number of data items and can have a bad impact on system scalability. Some tunable techniques to reduce its size can be used but they build on the assumption that an accurate real time clock exists to produce timestamp values, and transactions execute for relatively short periods of time.

The technique just described is said to be *aggressive* (we could say optimistic as well) in that it schedules transactions as soon as they arrive. There are also *conservative* variations which introduce a certain delay to reduce the conflict probability and so the number of rejections. We can build a conservative scheduler that never rejects operations if every transactions is able to predeclare its readset and its writeset.

2.2.2.2 Serialisation graph testing (SGT)

This method uses a *Stored Serialisation Graph* (SSG) to discover if a transaction that is to be submitted can create a cycle, leading to a non-serializable (non-SR) execution. If that happens, the transaction is rejected, that is, aborted. The SSG includes all transactions, but mechanisms to delete information can be used to reduce the complexity of the graph. Also for SGT there exists a conservative variation and an aggressive one as well.

[7] In case of a classic DBMS model between the scheduler and the data manager as explained in [79].

2.2.2.3 Certifiers

The approach followed by certifiers is to schedule operations as soon as they arrive and from time to time checking if all goes well. If so, a certification is done and the scheduling goes on, otherwise they must abort certain transactions. Obviously, not to commit transactions involved in non-SR executions, they must check at least before every commit operation. Thus, certifiers are also known as *optimistic schedulers*. There exist certifiers based on the 2PL idea and on the SGT as well. In distributed systems, the certification process is done only when a unanimous decision is reached by the local certifiers involved that must exchange their vote[8]. We discuss distributed databases later in this chapter.

2.2.2.4 Integrated schedulers

Integrated schedulers decompose the problem of concurrency control into two sub-problems, the *read-write synchronisation*[9] and the *write-write synchronisation*. Then, they usually deal with the two sub-problems using 2PL, TO or SGT consistently. There can be *pure schedulers*, which use the same mechanism for both sub-problems, and *mixed schedulers*, which use different approaches.

It is worth noting that each mechanism used for a sub-problem, must be redefined in order to reflect the new definition of *conflicting operations*; in fact, in read-write synchronisation, two writes on the same data item are not considered conflicting. In integrated schedulers the hardest part is to ensure the compatibility between the solution to the two sub-problems, as we can see from the TWR rule that follows. In general the problem is that of ensuring that if $SG_{rw}(H)$ (serialisation graph for the history made of conflicting read-write operations) and $SG_{ww}(H)$ (serialisation graph for the history made of conflicting write-write operations) are each acyclic then also their union $SG(H)$ is acyclic.

2.2.2.4.1 Thomas' write rule (TWR) The TWR never delays or rejects any operations. In case of a TO w-w synchronisation if it receives a $w_i(x)$ after it has already scheduled $w_j(x)$ and $ts(T_i) > ts(T_j)$ (ts is the timestamp) it simply ignores the $w_i(x)$ but it reports its successful completion. The key fact to prove its correctness follows from the fact that it is a w-w synchroniser, so the problem of r-w conflict is not its business but it will be solved by the r-w synchronizer.

[8] The problem of disconnected nodes is not considered so far.

[9] With r-w we also consider the symmetric problem w-r.

The TWR can be used as an optimisation of the basic TO mechanism, where the TWR is used for the w-w part and TO or strict 2PL for the w-r one.

2.2.2.5 Multiversion concurrency control (MV)

In MV schedulers every write operation produces a new version of a data item and the old copy is not thrown away. In this way some reads that should have been rejected (because their value has been overwritten) can be executed normally using an old copy. Keeping old copies is not a big effort since recovery mechanisms already keep old values of data overwritten by active transactions. Every write produces a new version of a data item. Old versions can be purged as soon as they become useless (with respect to active transactions). This process is hidden to the database user, so he/she still acts as if there were a unique copy of a data item. This method is used to improve the performance by rejecting operations less frequently.

A different scenario is where users want to explicitly access old versions; the MV mechanism described in this section is not suitable for those kind of applications.

2.3 Distributed Databases, concepts and consistency mechanisms

In [75] a distributed database (DDB) is defined as "a collection of multiple, logically interrelated databases distributed over a computer network". Distributed databases are often used within organisations that have offices in several towns or countries. In such cases, stored information is *fragmented*, that is, partitioned among the different offices. Data related to a given office can be stored at the same office where they will be more frequently accessed. Fragmentation can be done *horizontally* or *vertically*. Horizontal fragmentation separates rows, while vertical fragmentation separates columns and tables. The DBMS responsible for the management of a DDB that uses fragmentation should provide a way to hide this fragmentation to the end users of the database. Queries involving different fragments should be sent to the DBMS the same way a query on a single fragment is sent.

A DDB can implement also some sort of *replication*. Copies of the same data can be stored at multiple locations to achieve fault tolerance and speed up the data access. Also the replication process should be made transparent to end users. A user should ask the DBMS for the data without providing any information about its actual location; the user should not even know that there could be more than a single copy of a data item.

Distributed databases are inherently more complex than a central stand alone database. The design of the database is different, the query processing is different, concurrency control now has to span different locations, and security and reliability issues related to the use of a network to connect the databases must be considered. When replicated, database replicas must be kept synchronised.

When dealing with a distributed system, like a DDB, three important properties can be considered: autonomy, distribution and heterogeneity. Autonomy deals with the distribution of control (management, consistency and query processing), not data. Federated databases (like in IBM Federated Database Technology [73]), clustered databases (like in Oracle Real Application Cluster [33]) and replicated databases (provided in different solutions by several vendors, we will study in particular Oracle Streams technology in Section 3.5) are some specific types of distributed databases that mainly provide different levels of autonomy to each location. Different degrees of autonomy are possible for distributed databases, and the choice is sometimes related to the distribution of data. Distribution regards the distribution of data, fragmentation and replication. Heterogeneity may occur at different levels: hardware, operating system, database vendor and DBMS configuration. Among them, database vendor differences are the most difficult to cope with, since they often involve the use of different SQL support and different interfaces for data access.

Concurrency control and reliability in DDB can be separately studied and this is what we do in the next sections.

2.3.1 Concurrency Control in DDBMS

Concurrency control in distributed databases deals with the isolation and consistency properties of transactions. Dealing with distributed concurrency control, one can assume that the distributed system does not experience any fault; this strong assumption will be discussed in Section 2.3.2. As we are going to see, distributed concurrency control techniques are simple or more complex extensions of the techniques used for concurrency control in centralised databases, discussed in Section 2.2.

2.3.1.1 Primary site 2PL

In Section 2.2.1 we reviewed the concepts of the two-phase locking algorithm to manage the concurrency control within a single database. For distributed (replicated or partitioned) databases, a simple extension to the classic 2PL algorithm, known as *primary site 2PL* [77] can be applied. It consists in delegating the lock management

to a single database only. The other databases will contact the lock manager of the primary site whenever they need a lock operation.

2.3.1.2 Distributed 2PL (D2PL)

When lock managers are available at each site of a distributed database system, an extension of the 2PL algorithm, called *distributed 2PL (D2PL)*, can be used. In replicated environments, the D2PL implements a ROWA (Read One Write All) protocol, where a single replica can be contacted for a read operation but all the replicas must be available for a write operation. If the database is not replicated, the D2PL becomes the primary site 2PL.

Unreliable links and disconnected sites are an issue when using 2PL for replicated databases. To be able to complete a transaction when just a subset of all the replicas is available, other algorithms have been proposed (see for example [78]).

2.3.2 Reliability in DDBMS

The reliability problem for distributed databases includes the specification of commitment, termination and recovery protocols. Commitment protocols ensure the atomicity property (see Section 2.1) of distributed transactions. Termination protocols are used to allow the termination of a distributed transaction even when one of the participating sites is down, while recovery protocols deal with the recovery of the database that experienced the fault. In the next sections we briefly review the most important concepts of these protocols which will help the discussion in the rest of the thesis.

2.3.2.1 Two-Phase Commit Protocol

The Two-Phase Commit protocol (2PC) is used to ensure atomic commitment in distributed transactions, that is when data updates need to occur simultaneously at multiple databases within a distributed system; the process is synchronous, thus its application through unreliable links or with faulty nodes presents some difficulties. It has been used since the 1980s in critical sectors like airline reservation and banking applications. Different implementations and variations of this protocol exist; a simple overview of the main steps involved in the classical 2PC protocol is:

Prepare phase the initiating database sends a query to all participants requesting that they will all either commit or abort the transaction. Each participant executes the query without committing it, and send an acknowledgement to the initiator.

Commit phase if the initiator receives all the acknowledgements from all the participants, then it sends back a request to commit the transaction. If all the participants successfully execute the commit operation, then the initiator closes the circle committing its transaction. If the initiator does not receive a positive acknowledgement of his request from all the participants, then it sends a global abort message to everybody and the transaction does not happen in any of the databases.

An interesting aspect of this protocol is that an abort decision acts as a veto, that is the abort decision can be unilateral. Thus, this version of the protocol can only lead to a global commit only in absence of failures. We should also note that, in this version of the 2PC protocol, the participants do not communicate among themselves, but only with the initiator. This is the reason why this protocol is also called *centralised 2PC*.

Variations of the 2PC protocol exist depending on the communication pattern and the network topology. For networks that do not have broadcasting capabilities, a variation of the 2PC protocol called *linear 2PC* is possible. In this case the prepare message and the acknowledge messages can flow from node to node in a linear way [75]. Distributed 2PC is another variation and involves messages from the initiator to all the participants in the first phase, but then each participant sends its decision to all the other participant and the initiator [75]. Thus, every site is able to decide whether to commit or abort the transaction by its own, without the need of a global commit or abort message.

Termination protocols ensure that a transaction reach a final state even in the presence of a site failure. In this case a timeout procedure is used, both at the initiator and at the participant site. Timeouts procedures must be tuned considering the application and the network topology, and a trade-off between efficiency and correctness must be found.

2.3.2.2 Three-Phase Commit Protocol

The 2PC algorithm is blocking in the sense that when a failure at the initiator site happens, participant sites have to wait for its recovery before taking a final decision on the transaction. To overcome this problem, the three-phase commit protocol has been developed [51]. We do not go through the details of this protocol since it does not add concepts useful to the scope of this thesis. It is however important to point out that the 3PC protocol is not able to deal with partitioned networks. In case of network partitioning, partitions will continue to process operations independently.

2.3.3 Lazy approaches in replicated databases

In [59] it was found that blocking protocols have serious performance degradation when implemented on wide area replicated databases; deadlocks increase as the cube of the number of sites and as the fourth power of transaction size. Lazy approaches try to offer better performance removing the atomic commitment; once a transaction has committed at the originating site, it eventually commits independently at replicated sites. In [88], a single-master (or primary site) approach is described; it is able to tolerate site failures but not network partitions. Another lazy protocol that is not based on a single-master approach, is described in [60]. Its approach is what is generally called *write-anywhere*, since a client can perform write operations on all the replicated databases. Vector clocks [64] are used to order operations and each site stores enough information to have a complete picture of the events in the overall system. This protocol avoids global deadlocks and reduces delays caused by locking. In order to commit an operation, all sites must be eventually available, but the protocol can be extended using a quorum system to resolve commit decision among a subset of available sites.

Chapter 3

Replication and consistency in wide area data stores

In distributed systems data are often replicated to several sites in order to improve data availability and fault tolerance, but also to provide users with fast data access. Data are in fact replicated close to the users that need to access them. When replicas can be modified, we need a mechanism to keep them consistent; this is the goal of a synchronisation mechanism, that can be achieved using different techniques. In this chapter we study different synchronisation techniques and their application to several application environments.

In Section 3.1 we review the main concepts of data replication, stressing the difference between static replication and replica synchronisation. In Section 3.2 different synchronisation techniques are studied. Since we are especially interested in synchronisation over wide area networks, we focus on *optimistic replication*, where replicas are kept consistent in a "relaxed way". In the rest of the chapter we review some practical applications of these concepts. In Section 3.3 we present the SWARM middleware and its composable consistency model, while in Section 3.4 we present the TACT project. We conclude the chapter with a section about Oracle Streams, the Oracle solution for the synchronisation of Oracle databases, that, as we will see in Chapter 4, is extensively used at CERN to replicate Physics data.

3.1 Data Replication

Data replication is a well known strategy to improve performance and reliability of distributed computing platforms. When the users of a system are distributed over a wide area network (WAN), keeping data at a single location can affect data access in three ways:

Latency The data access time varies with the distance and link bandwidth of the user from the data storage, and it is subject to network problems related to the WAN environment.

Availability Having a single data storage site is a risk for critical applications: when storage is temporarily unavailable (for faults or maintenance reasons), or the storage site is not reachable due to network problems, users do not have access to data.

Congestion The single data storage site must sustain a potentially high number of users requests; the hardware used for data storage can be very expensive or fail to satisfy user requests.

For these reasons, in many distributed environments data are replicated, i.e., copied, at different locations. In Figure 3.1 an environment with a single data provider is shown. In Figure 3.2 instead, we show the same system with the data storage replicated in three locations.

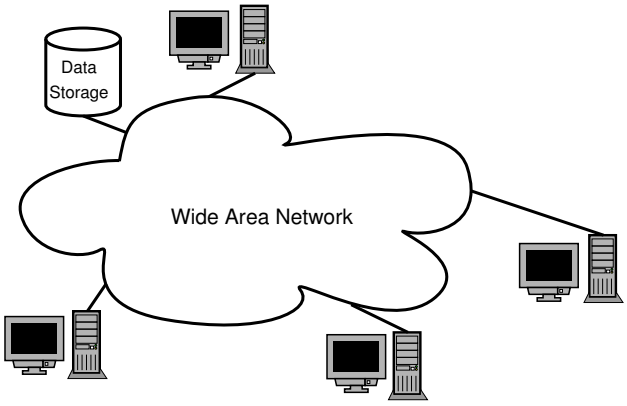

Figure 3.1: Distributed environment with a single data storage

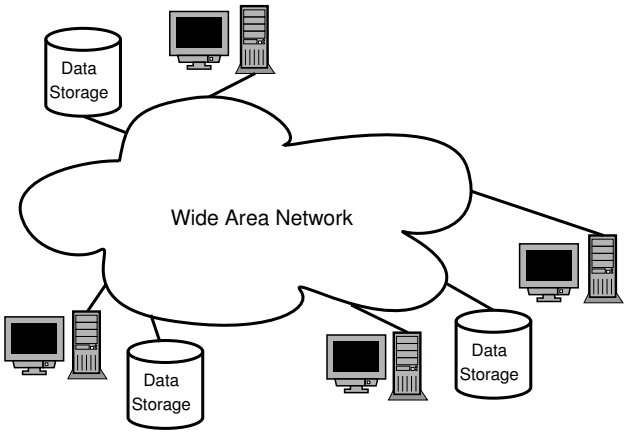

Figure 3.2: Distributed environment with replicated data

In the system that replicates the data store we do not have a single point of failure for data storage: if a storage component fails, users can still have access to the information through the other two replicas. Users also have data closer to their location, which speeds up data access. Besides, since the load on the storage devices is shared among the replicas, the storage devices can be built using less expensive hardware or, in any case, can better sustain user requests. To summarise, with data replication we achieve: *fault tolerance*, *fast data access* and *load distribution*.

Unfortunately, the benefits before mentioned come at a price. Having different replicas of a same data item, we need a way to locate them, hence saving their location and some metadata information. As we will see in Chapter 5, *replica catalogues* are used to this purpose, together with a well defined naming scheme. Replicas must also be created and placed according to the system configuration, or based on some dynamic criteria. For example, a replica could be created and saved close to a user just for the time the user needs that data, and then removed from the system. This is what is called *dynamic replication*, or sometimes *replica optimisation*. Moreover, we have to face problems of replica *coherence*, and *synchronisation* when replicas can be independently modified by users. In particular, replica coherence and replica synchronisation can be two faces of a more general problem, that of maintaining *replica consistency*.

Generally speaking, replica consistency means that replicated data have the same content. Then, since contents can diverge for different reasons, we distinguish be-

tween the two problems:

- replica coherence: even when replicas are read-only, from the point of view of a user application, replica coherence can be broken when one of the replicas gets corrupted, either for hardware or software problems. In this case the content of the corrupted replica is different from the one of the other replicas. Replica coherence must be checked also when a new replica is introduced into the system: the content of the replica should be checked in order to ensure that it is a real copy of the replicas already present in the system. It is worth stressing the fact that replica coherence must be checked also in case of read-only replicas.

- replica synchronisation: synchronising replicas in only needed when replicas are modifiable. When a replica is modified by a user, the other replicas become stale. In order to re-establish consistency among replicas we need to propagate the modification done on the first replica to the others, in a process that is usually called *update propagation*, the main part of a replica synchronisation process.

Replica coherence can be enforced with periodic checks on replicas, and controlling the creation of new replicas. Replica synchronisation is more complex to deal with, and different approaches exist. Before studying these approaches in Section 3.2, we clarify, in the next section, the difference between replication and synchronisation, which is sometimes a source of misunderstandings.

3.1.1 Snapshot or Static Replication

Often, the terms “replication” and “synchronisation” are used with the same meaning. However, they are different in that they involve two different tasks, that frequently are complementary. Replication is the task of replicating, i.e., copying, a data item at different locations, generating new replicas. Synchronisation is the task of keeping these replicas consistent, when they are modifiable. A synchronisation system does not create replicas, but works on existing ones, therefore relying on a replication system. However, a replication system can work with or without a synchronisation system. When replicas are read-only, or when no consistency guarantees are required by the users of the replicated system, a synchronisation system is not needed. In these cases we often refer to *static* or *snapshot* replication.

In databases, snapshot replication provides a replica of a database, or part of it, as it was in a particular point in time. When data changes are infrequent and the size of the data is not very large, the replication process can be performed periodically.

Snapshot replication does not propagate changes, but database snapshots, therefore the consistency of the replicas is not guaranteed when the main database is modified. In certain use cases, as when storing history information, a replicated database could be configured with a master site containing all the up-to-date information and replicas containing data belonging to different non-overlapping periods of time. This case is depicted in Figure 3.3, where the main database holds history information from the year 2000 to the present, and the replicas hold data in intervals of two years extent. This is a useful way of off-loading information especially when the load on

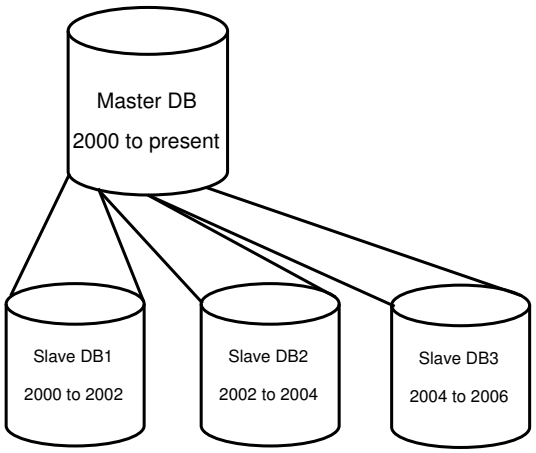

Figure 3.3: Scenario for a Snapshot replication, static replication of history data

the database is mostly made of read operations. In this case queries can be redirected to one of the slave replicas depending on the information the user needs. This redirection can be transparent to the user or not, and can rely on a central or distributed catalogue to find out the location of the database to query.

Snapshot replication is implemented in many relational databases, among which MS SQL Server [38] and Oracle [35], where snapshots are known as *materialized views*.

Cross-technology replication tools exist: Enhydra Octopus [10] is a Java-based Extraction, Transformation, and Loading (ETL) tool that allows data to be replicated statically using different database technologies. Octopus has been tested in the ATLAS experiment (see Section 4.2) for replicating, statically, geometry and tag databases [28].

3.2 Replica synchronisation techniques

While replica coherence can be enforced with proper checks when a replica is created and added to the system, or with periodic checks on all the replicas, replica synchronisation presents more issues and different protocols for update propagation exist. A first classification can be done between *pessimistic* and *optimistic* techniques.

Pessimistic techniques are often used in replicated databases, in order to provide *single-copy consistency* [47] (also know as *one-copy serializability*). Pessimistic techniques can be used in a LAN environment, where the system is made of few homogeneous copies. These techniques give users the illusion of having a single, highly available copy. As we saw in Section 2.3, they are usually implemented using locking techniques. During the update of a replica, the access to all the other replicas is blocked until the update has been correctly applied to the first replica and propagated to the others. For this reason they are also called *synchronous*. Pessimistic techniques are not feasible when replicas are connected through unreliable links, through links with low latency, or when replicas are subject to failures. The update propagation would have to delay the update of a replica, and, even worse, could be blocked indefinitely or rejected in case of network partitions. Moreover, the performance of pessimistic techniques decreases when the number of replicas is high and updates are frequent [59]. These are the reasons why in many distributed applications optimistic replication is often preferred.

Optimistic replication[1], or asynchronous replication, is the term used when, in a replicated environment, the update of a replica is propagated to the other replicas in a relaxed way, that is, after the commit of the original transaction. Such a way of synchronising replicas is called optimistic because the access to the replicas is not blocked when one of them is being updated, and this could cause stale reads for a limited amount of time. In these cases we also talk about *relaxed consistency* [43], or lazy consistency. Delaying the propagation phase helps to speed up the write access even in the presence of slow and unreliable links. As for pessimistic replication, however, the overall performance decreases and the system gets complicated when all the copies are writable, and write operations are frequent. Coordination among write operations is fundamental for the effectiveness of an optimistically replicated system. In the next section we see different approaches that can be followed for implementing an optimistic replication mechanism.

[1] When the term "replication" is used with optimistic, or pessimistic, that characterises a synchronisation technique, it is implicit that we talk about replica synchronisation, and not about static replication.

3.2.1 Optimistic replication

Different types of optimistic replication can be implemented, and in this section we see the main criteria to distinguish them. A fairly simple way to do it is through three simple questions, that we call the *3 w's questionnaire*:

1. *Where* can an update be done?
2. *What* is transferred as an update?
3. *Who* transfers an update?

The answers define the system as it is explained in the following sections.

3.2.1.1 Single-master vs multi-master

The first question of the 3 w's questionnaire helps to separate single-master systems from multi-master systems, which is the main classification for optimistic techniques. We call "master" (or primary replica) a replica that can be updated by end users and that always has the most recent version of a data item[2]. In single-master systems only one replica is designated to act as master: this replica is the only one that can be updated by users. The other replicas are read-only slaves (or secondary replicas), and cannot be updated by users but only by the replica synchronisation system. Single-master systems are optimal when updates are not frequent, and data access is mainly for reading. In this case read operations are faster than in a non-replicated environment, the system is more fault tolerant for read operations and the read load on the replicas is balanced. However, these improvements do not apply to write operations; they still must be issued on the single master replica, thus the master replica can become a bottleneck for writing, and a single point of failure as well. For what concerns the single point of failure, the effect can be reduced by implementing an election algorithm among the slave replicas in case the master is unavailable. Slave replicas can elect a new master, and the old master will become a slave replica as soon as it is available.

Single-master systems are fairly simple to implement, but the update propagation phase must be properly designed when the slave replicas can be temporarily unavailable. In these cases a *quorum* system can be implemented to perform an update propagation only when a quorum among the slave replicas is reached, and the remaining

[2] In case of multi-master systems master replicas can temporarily diverge for the time they need to exchange the updates and solve potential conflicts.

replicas can be updated later on. The performance of the update propagation phase is fundamental in order to decrease the probability of having stale reads on slave replicas. As we will see in Chapter 6 and Chapter 7 the Replica Consistency Service we developed uses a single-master approach, with a quorum system and special provisions to manage disconnected sites. A detailed performance analysis has been done in order to have an estimate of the lazyness of the system and of the factors that affect it more.

In multi-master systems there are more master replicas that can be used for both reading and writing, the extreme case being the one where all the replicas are master, also known as *update anywhere* solution. Thus, unlike in single-master systems, performance increases for both read and write operations, and failures of a master replica are better tolerated by the system since a user can write on another master, without the need of election algorithms. The main drawback of multi-master systems is that concurrent writes on different masters need to be carefully managed and possible conflicts resolved. An update conflict happens when two users update the same data item at different master replicas. Usually masters exchange their update in the background, and the way they resolve conflicts is very application-specific. In some applications, where write operations are "append" operations, or where the content of different updates can be easily merged, multi-master solutions are very effective. On the other hand, when the semantics of the application complicates the update resolution phase and when updates are frequent, multi-master systems can become too difficult to manage and can provide no solutions for lost updates.

3.2.1.2 Content-transfer vs log-based

The second question of the 3 w's questionnaire raises the point of defining what an update is. In some systems, where replicas are for example flat files, *content transfer* mechanisms are used, meaning that an update can be either the whole new version of the file, or just a piece of it, usually the binary difference between the old and the new version. For small changes, the whole replica can be used to replace the old versions (total file replacement). When changes are large, we can improve the propagation phase by propagating just the difference between the new and the old version: a difference extraction/application tool must be available for the type of data managed. In a Unix environment the `diff` and `patch` command can be used to this purpose.

In other cases, for example with structured data as in relational databases, updates can be described by sets of operations (i.e., transactions) and we can propagate the updates just propagating the operations that generated them. That can be much faster

than propagating all the rows involved in an update. In these cases we talk about *log-based* mechanisms. As we will see in Chapter 6, our Replica Consistency Service uses a log-based method for synchronising databases, and a total file replacement (hence, content transfer) system for flat files.

3.2.1.3 Push-based vs Pull-based

The third question of the 3 w's questionnaire helps to discriminate between push-based systems and pull-based ones. In push-based systems the replica that has been updated is responsible for triggering and performing an update propagation. In pull-based systems it is up to each slave replica to retrieve the updates and synchronise its content with the one of the up-to-date replicas. The choice of using one or the other method depends on several factors, like the frequency of read and write operations and the availability of secondary replicas. For example, when updates are frequent on the master replica but read operations on secondary replicas are rare, it is convenient to update the content of the secondary replicas with a pull-based mechanism, only when needed, that is when the read operation is requested. On the other hand, when updates are rare, and read operations on slave replicas are frequent, it is convenient to immediately update the secondary replicas with a push mechanism as soon as the updates are available on the master replicas.

3.2.1.4 Consistency guarantees

Sometimes, synchronisation mechanisms are grouped based on the consistency guarantees that they offer. For example, the weakest guarantee that each synchronisation system must provide in order to be considered "correct" from a functional point of view, is *eventual consistency*. This guarantee states that the content of the replicas, when no new updates are issued, will be *eventually* the same [89]. In other words, a replica may remain inconsistent for some time. Many applications can work well with such a weak guarantee, others may instead require that users will never read data that are older than a fixed amount of time. To provide such a guarantee we need a way to estimate replica divergence, and prohibit the access to replicas when the divergence exceeds a given threshold. In a single-master system, a limit on the staleness of a replica can be fixed by using a pull based system where secondary replicas synchronise with the master replica periodically, with a period less than the maximum allowed staleness.

In [63] four *per-session guarantees* are proposed as a way to provide users with a view of the database that is consistent with their own actions at different levels of

complexity. For example, a user may require a level of consistency in order to be sure that a read operation always sees previous writes done by the same user (*read your write*, RYW), even if different replicas are used for reading and writing.

3.2.1.5 Conflict management in multi-master systems

As we discussed in Section 3.2.1.1, multi-master systems can improve the performance of both read and write operations. However, they must be carefully planned in order to avoid conflicts between concurrent writes on the same data item done at different replicas. Conflict management is a very application specific topic, and can be dealt with in two ways: avoiding the conflicts, or detecting and resolving them. Conflict avoidance can be obtained with locking techniques, as done in pessimistic replication. In optimistic replication, where locking not used, conflicts may happen and need to be detected and resolved. A straightforward solution uses the Thomas's write rule (see Section 2.2.2.4), where write operations are applied in their timestamp order.

For serialising operations scheduled at different replicas, and detecting conflicts, the most known method is using *vector clocks*. A vector clock is a data structure that is used to propagate synchronisation information in a distributed environment. It is usually implemented as an array of M elements, where M is the number of master replicas. The array contains timestamp values; when site i has the timestamp value a in the $j-th$ element of its vector clock, it means that it has received all the updates from site j with timestamp up to a. The size of this data structure is the main concern when vector clocks are used in environments with many replicas, and can impose a limit on the system scalability.

Once detected, conflict must be resolved. Manual and automatic conflict resolution mechanisms exist. Manual conflict resolution detects conflicts and presents to the user two versions of the data; it is up to the user then to resolve the conflict either merging the content of the new versions or discarding one of them. This is the system used in code repositories like CVS [55]. Automatic conflict resolution is performed by specific procedures that are applications specific. In replicated file systems different procedures are used to resolve conflicts depending on the type of file that experienced the conflict. For example, when the file is a compiled file, the conflict can be resolved by recompiling it again from its source.

3.3 SWARM and its composable consistency model

In this section we present SWARM, a middleware for wide area replicated data stores that implements a consistency model for peer-to-peer systems called *composable consistency* [84]. SWARM architecture comprises a collection of servers (*SWARM Servers*) that are put on top of distributed data stores, and provide a session-oriented file interface that can be used by its client applications to create and remove files, open a file session with specific consistency options, read and write file blocks and close the session.

In SWARM, a session[3] is the unit used to define the consistency behaviour. In other words, the consistency of the operations made on a file is managed during the opening and the closing of a file. For example, when opening a file, the user can choose whether to retrieve the most up-to-date copy of the file or deal with the possibility of having a stale reads. In the same way, when closing a file, a user can choose whether to propagate immediately the updates done or not. In order to use SWARM an application has to link to the SWARM client library and use the file interface provided with it. When a user wants to access a file, he can work on a local replica, and specify the consistency behaviour of the current session; SWARM servers will then communicate to implement that specific behaviour.

The composable consistency model can be applied to applications structured in a peer-to-peer paradigm, where each site holds a portion of the application state and caches other site's information. A small set of primitive options are used by applications to build a consistency model which is appropriate to their data sharing needs.

3.3.1 Composable Consistency options

Composable Consistency (CC) uses five basic options to define the consistency requirement of an application, at the session level:

- *Concurrency*: it defines whether, within the current session, read and write accesses to a replica must be done in a concurrent or exclusive way with respect to the other replicas. When the exclusive way is chosen, the access to the other replicas of the same file is blocked until the current session is finished.

- *Replica Synchronisation*: it defines the replica staleness that can be tolerated by the current session.

[3] In this context a session is a set of operations made on a file, that start with an `open(file)` and end with a `close(file)`.

- *Failure Handling*: it defines the update propagation behaviour of the current session when some replicas are unavailable or have poor connectivity.
- *Update Visibility*: it defines when new updates generated by the current session must be made visible to the other replicas.
- *View Isolation*: it defines when updates generated remotely must be made visible to the current session.

These five options are considered enough to define the consistency behaviour of a wide range of peer-to-peer applications. They are considered orthogonal, even if they are not completely independent. Each option has a predefined set of values that application programmers can use to build the model more suitable for their application. Options also have a default value, and some predefined combinations of option values are also provided, grouped into well known consistency behaviours. A user can choose one of those predefined combinations, or refine its consistency model selecting the most appropriate value for each option.

3.3.2 Replication in SWARM

SWARM servers that hold a particular replica, organise themselves in a replica hierarchy: the root (or home) node is the one where the file was initially created, and its server is the one that coordinates all the consistency operations on that file replicas. However, for each file, there can be more than one home node, to increase the fault tolerance; when a single home node is present, an election algorithm is used to elect a new home node in case the first fails. When a server wants to have a replica of a certain file, it contacts the home node of that file and obtains a copy becoming automatically child of the home node for what concerns that file. Network links are constantly monitored by the servers in order to perform some optimisation tasks; for example, when looking for a home node, a server, in case more home nodes are present, can choose the one that is "closest" to it regarding the network speed. Each server has a limited fanout: when a home node is saturated, one of its children can be used as home node for new replicas. In this way the communication overhead is distributed over the replica hierarchy.

3.3.3 Implementation of composable consistency

To implement the composable consistency model outlined in Section 3.3.1, each SWARM server has a consistency module, that performs specific operations each

time a user opens, accesses and closes a file. The consistency module implements the required consistency model by communicating with the other servers. The data structure used by each consistency module to save consistency information about replicas is called *privilege vector* (PV). Privilege vectors are exchanged by peer consistency modules in order to learn of each other's PVs. Thus, each module knows when it can grant a file access to a client without contacting its peer modules, or when more coordination is needed. We do not go through the details of the implementation of the different models since the details of this system are out of our scope. The SWARM architecture and its composable consistency model has been evaluated building a SWARM-based file system on a cluster of PCs using the Emulab Network Testbed [9] to emulate WAN topologies among the clustered PCs [83].

3.4 TACT

In this section we outline the TACT (Tunable Availability and Consistency Tradeoffs) [68] toolkit, a middleware layer that mediates read/write accesses to underlying data stores in order to enforce some tunable consistency requirements. TACT provides different consistency models that are logically placed between strict consistency and eventual consistency, using a set of metrics to define the consistency requirements of a distributed application. Using TACT the tradeoff between performance and consistency can be quantified for some specific application use cases, and the consistency requirements can be dynamically varied in order to cope with network conditions.

The implementation of TACT has the same logical structure of the SWARM servers, that sit on top of a replicated data store and provide a file interface to access data. A TACT server can limit the staleness of a replica by intercepting and in case blocking new updates when the number of uncommitted operations exceeds a user-specified threshold. In other words, it blocks any attempt of modifying a file when the file is older than a certain threshold.

In TACT, an application can define its requirements in terms of consistency by defining a *conit*, that is, a logical unit of consistency. Different granularity levels can be specified by changing the definition of a conit. An example of conit could be a file on a filesystem, a table inside a database and so on. Then, for each conit, a consistency value is quantified as a three-dimensional vector of: numerical error, order error, staleness. Numerical error gives a measure of unseen writes, order error measures the number of out of order writes on a given replica, and staleness bounds the difference in time between the present and the time of the oldest write not seen

locally. When the three values are zero the consistency value is the same that one would obtain in a synchronous system. Using these three measurements, the middleware can enforce consistency bounds among replicas.

The details of the implementation of the update propagation algorithm can be found in [68], where the TACT toolkit has been applied and evaluated for three different applications: a bulletin board, an airline reservation system and a system to enforce quality of service in web servers.

3.5 Oracle Streams

Oracle Streams [34] is an Oracle product to achieve high speed replication and data sharing among widely distributed sites. It was introduced as a new major feature in the Oracle release 9.2. Before Streams, replication of Oracle databases was still possible using snapshot replication (see Section 3.1.1). As depicted in Figure 3.4, the Oracle Streams working model is made of three phases: capture, staging and consumption.

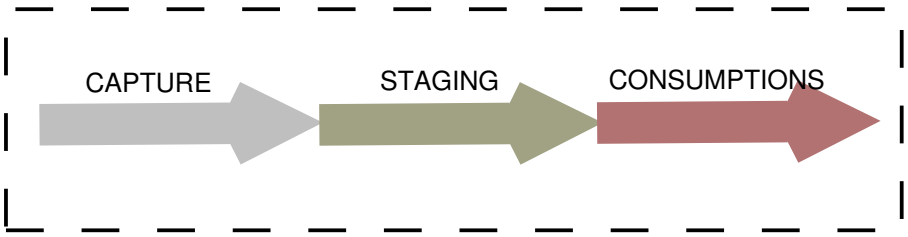

Figure 3.4: High-level view of Oracle Streams

SQL DML (Data Manipulation Language) and DDL (Data Definition Language) changes are captured at the source database, staged into a queue, and consumed at destination sites. Besides DML and DDL changes, Oracle Streams is also able to capture user defined events.

At the source database, log files are read in order to extract changes made to a table, schema or another database object. These changes are then translated into specific update units, called *Logical Change Records* (LCRs), enqueued in specific queues, propagated and applied to databases at destination sites.

LCRs are enqueued using an Oracle specific mechanism called *Advanced Queueing* (AQ). The update propagation happens between a source and some destination queues created with AQ, using a publish-subscribe mechanisms.

All the Oracle Streams activities included in the three main phases (capture, staging, consumptions) are performed by three main processes: *Capture*, *Propagation* and *Apply*. As the name suggests, the Capture process is used at the source database to capture specific events, usually DML and DDL queries. The Propagation process is used to transfer data from a source to a destination queue, and the Apply process to extract events from a destination queue and apply them to the destination database. The events flow is shown in Figure 3.5.

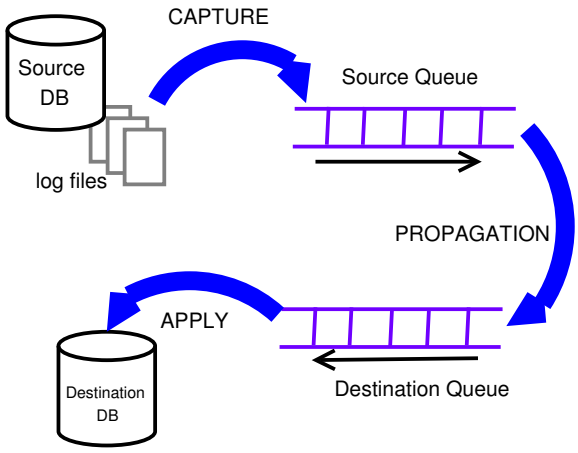

Figure 3.5: Architecture and data flow in Oracle Streams

In Oracle every change done on a database is recorded into the *redo log* files (on-line and archived, as explained in Section 6.4.3.8). Entries in a redo log file are converted into LCRs events (or captured events) by the Capture process and then stored into the source database queue. An LCR is implemented as a specific Oracle object with a specific *System Change Number* (SCN) and can be a of two types: Row LCR or a DDL LCR. In case of Row LCR, DML changes are stored recording the old (*before image*) and the new value of a row, together with additional metadata (database name, schema, session id, etc.). A DDL LCR instead contains the DDL statement that modified the database, for example a `create table` statement.

The Capture process usually runs on the source database. However, in order to improve the performance and reduce the load on the source database, the Capture process can also run on a separate database, called the *downstream database*. In this case, the Capture process reads the archived redo log files that are shipped from

the source database to the downstream database. The downstream database is just another Oracle database where the Capture process and the associated queues are created. In the downstream database the Capture process reads the archived log files while, when the Capture process runs on the source database, it can read either the on-line or the archived redo log files. This means that, when using a downstream database, the synchronisation process is delayed because the Capture process has to wait for the archived logs to be written and copied to the downstream database[4].

Captured events can be stored in more than one queue, at the source database, if necessary. In this case a Capture process must be defined for each queue. The Oracle LogMiner utility is used by the Capture process to scan the redo log files; when more than one Capture process is defined, each Capture process works with a separate LogMiner session. For the LogMiner to function properly, *supplemental logging* must be enabled on the source database. This means that the logging activity is enhanced and extra data are written into the redo log files. Another important features is the possibility to define rules on the Capture activity, so that LCRs are enqueued or discarded basing on defined rules.

Captured events are dequeued from the source or downstream database queue and enqueued into the destination database queue by the Propagation process. A queue at a source database can provide data for multiple queues at some destination sites. In this case a Propagation process must be defined for each destination site. In other words, a Propagation process cannot propagate data to multiple destinations or from multiple sources. We can define rules also within a Propagation process. When a rule is evaluated to "true", the event is propagated, otherwise it is discarded.

When LCRs go into the destination queue, the Apply process can either apply them to the destination database, or pass them to some specific handlers, that are user defined procedures.

It is worth noting that, when a table in created at the source database, it is replicated to the destination site. But what happens if a table containing some rows already existed at the source site, before setting up the Ctreams processes? How is the destination database synchronised with the source database? A starting point must be fixed in order to start correctly the Streams replication. This operation is called *instantiation* and can be achieved by copying the table with one of the import/export methods available in Oracle. After the copy, an SCN value is fixed as a starting point for the replication process.

As most of the database replication solutions, Oracle Streams can be used both

[4]In Oracle the archived log files are written only when an on-line redo log file is filled, and a switch of the on-line redo logs happens. In Section 6.4.3.8 more details are given.

for load sharing and fault tolerance.

3.5.1 Bi-directional synchronisation

Bi-directional replication is the term used when two database replicas are synchronised and changes can happen at both databases, that is when we have a multi-master configuration. The propagation of updates must occur in both directions. With Oracle Streams bi-directional replication can be implemented, but the management of this scenario can be very complex.

A first issue that must be considered is the cyclic propagation of updates. When two databases, namely DB1 and DB2, are synchronised with bi-directional replication, changes occurred at DB1 are captured and propagated to DB2. But, since there is a bi-directional replication, also the changes done in DB2 are captured, and should be propagated to DB1 as well. Obviously, changes generated at DB1 must not come back and applied again to DB1. To avoid the cyclic propagation of updates, Oracle Streams allow the use of *Streams Tags*. In this way, entries in the redo logs generated at DB1, can be labelled with a specific tag. These changes will be propagated and applied to DB2, but here the Capture process will have a specific rule to avoid capturing LCRs with that specific tag. Thus, only changes generated at DB2 will be captured and propagated to DB1.

The most difficult issue that must be solved in every bi-directional (or multi-master) synchronisation is conflict resolution, as already explained in Section 3.2.1.5. In the next section we see the approach followed by Oracle Streams to deal with conflicts.

3.5.1.1 Conflict resolution in Oracle Streams

Conflicts happen when, in a replication environment, more than one database can be updated at almost the same time.

A straightforward solution that is not valid for all the scenarios, is to separate the ownership of data. In this case, each database is responsible of updating directly part of the data, while the rest will be updated through the Streams processes. This solution is not always possible, and it depends on the semantics of the application.

Different types of conflicts are possible. Some of them can be avoided with some application design solution, others require specific conflict resolution policies.

The conflict that arises when an insertion with the same primary key (for example a sequence number) is done at all the master databases, can be solved using a different

primary key that includes a different number associated to each database, so that all new values generated at different sites are different.

In case of conflict associated with delete operations, we can transform a delete operation in an operation that simply marks a row for deletion. Then, only one site will be responsible for deleting marked rows.

When it is not possible to separate the ownership of data, or use some design solutions to avoid conflicts, conflicts must be explicitly detected and resolved. Oracle detects conflicts using the before image: when two changes are applied to the same row, at the same time, but at different sites, when they are propagated and applied, the before image is not the one expected, and both the databases experience an error. The Oracle default action would be to place these changes in an error queue, but the database administrator can set up a specific conflict resolution procedure to resolve conflicts before they end up in the error queue.

In Oracle Streams, some prebuilt conflict handlers are available, that implement one of the following policies: maximum, minimum, override and discard. The first two methods perform a comparison of the two values that caused the conflict and keep either the maximum or the minimum. Otherwise, a conflict handler can decide to override the present value, or to discard the new value. A database administrator can also choose to implement his own conflict handler (custom conflict handler), in the form of a PL/SQL procedure. A different conflict handler can be specified for each table. Moreover, in case both a prebuilt and a custom conflict handler exist for the same table, the custom handler has the precedence unless, inside its code, some specific options are set to redirect the control to the prebuilt handler.

Chapter 4

Requirements from Particle Physics

In this chapter we introduce the main application field that has driven this work, high energy Physics and the experiments performed at CERN. Although Grid computing is not exclusively tied to high energy Physics, this discipline has been the one that provided the necessary requirements and the one where this work has been developed. Particle Physics is a very complex discipline; in this chapter we only give a very informal introduction to some main concepts.

Within a particle accelerator, beams of protons are made to collide at very high energy. The collisions are analysed through particles detectors that generate large amounts of data for each collision. These data are recorded and analysed to discover the possible creation of new particles. With the ATLAS experiment and its model for data distribution, we describe one of the main scenarios where this work on replica consistency can be effectively applied.

The chapter is structured as follows. In Section 4.1 we introduce the particle accelerator in construction at CERN, and its goals. In Section 4.2 the ATLAS experiment and its computing model are presented, with a focus on the management of conditions data. Conditions data storage and distribution in the ATLAS experiment are further discussed in Section 4.3. In Section 4.4 the LCG-3D project is presented, and the techniques used to distribute data from Tier-0 to Tier-1 and Tier-2 sites are explained.

4.1 Particle Physics at CERN: the LHC experiments

Two questions are the main concern of particle physicists: what is the world made of? And, what holds it together? Physics research has been investigating for centuries in "fundamental particles", trying to discover the unit of matter. From water, fire, air and earth we passed to atoms, and then to nuclei (protons plus neutrons) and electrons. Then, they found out that protons and neutrons are made of quarks, and now they start suspecting that even quarks and electrons are not fundamental, that there is something smaller. They also discovered that for each particle, an antimatter particle exists. All the known particles, their behaviour and their interactions are explained in the Standard Model, the most complete explanation of the basic constituent of matter to date [76].

Particle accelerators have been extensively used by modern particle Physics to discover new particles and their properties. The *Large Hadron Collider* (LHC), in construction at CERN, will be the world's most powerful particle accelerator ever built. In its underground tunnel[1] two counter-rotating beams of protons, guided by powerful superconducting magnets, will circulate very close to the speed of the light and will be made to collide at four points where the detectors of the four LHC experiments (ALICE, ATLAS, CMS and LHCb) are placed.

Current accelerators increase more and more the energy of the circulating beams, since certain phenomena can happen only at extremely high energies. This is the reason why particle Physics is often referred to as high energy Physics[2].

More than a hundred nations have been participating in the design and the construction of the LHC in all its components, the tunnel, the detectors and the computing infrastructure that will be used to analyse data coming from the detectors.

Much of the effort of the LHC experiments is put into finding elementary particles and to explain the way they interact with each other. Dark energy, dark matter, antimatter and supersymmetry are some of the Physics topics where the LHC is expected to bring light. In order to do so, huge amounts of data coming from the detectors must be analysed, of the order of Petabytes (10^{15} bytes) every year.

The storage and the distributed analysis of this unprecedented amount of data requires a new computing infrastructure. To this purpose CERN is promoting the development of the Grid technology, that we discuss in details in Chapter 5. Grid computing is used by the LHC experiments to perform part of their data analysis

[1] Built 100 metres beneath the Franco-Swiss border, with a circumference of 27 kilometres, see Figure 4.1 and 4.2.

[2] In LHC, energies of 14 TeV will be reached during the collisions.

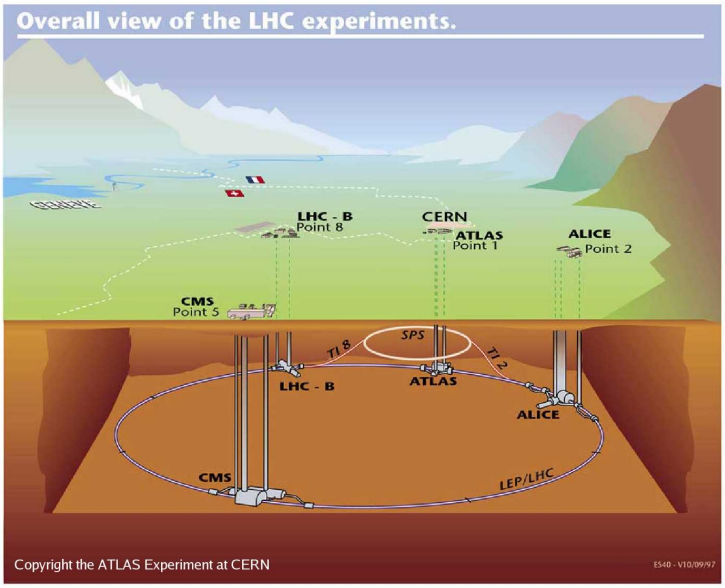

Figure 4.1: The Large Hadron Collider

tasks.

In the rest of this chapter we focus on the ATLAS experiment, since its computing model shows the need of employing the replication of heterogeneous databases which is one of the main topic of this thesis. It will also help to understand the main requirements that led to the use of Grid computing in high energy Physics experiments.

4.2 The ATLAS experiment

ATLAS is a high energy Physics (HEP) experiment that will explore the fundamental nature of matter and the basic forces that shape our universe. With a diameter of 25 metres, a length of 46 metres and a weight of about 7000 tons, the ATLAS detector (shown in Figure 4.3) is the largest among the four LHC detectors. It has a layered structure; each layer is made of different material and has a different goal. Particles created during a collision radiate in all directions and pass through one or more of these layers leaving behind them a “signature” that allows them to be identified. The results of a collision are then filtered so that only interesting *events* are recorded and

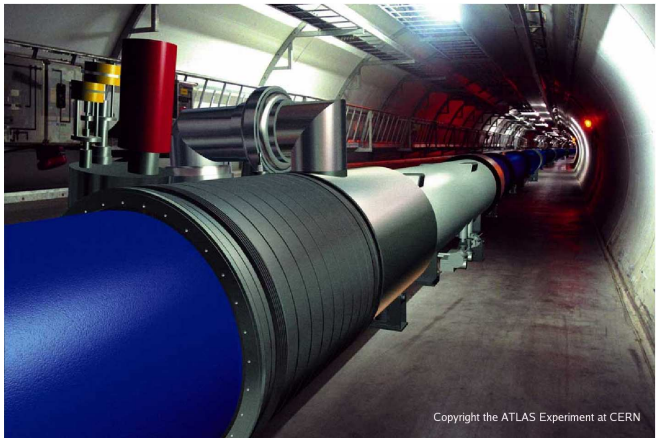

Figure 4.2: An inside view of the LHC tunnel

processed for making them suitable for Physics analyses. The ATLAS experiment will provide data to bring light in topics like *dark matter*, relation between *matter* and *antimatter* and proofs of the existence of the *Higgs particles*[3]. The experiment involves more than 1900 scientists from more than 160 universities and laboratories in 35 countries.

4.2.1 The ATLAS Computing Model

In the ATLAS detector collisions are expected to happen at a rate of about 40 MHz but interesting collisions, called *events*, that is collisions that may lead to an interesting configuration of outgoing particles, will be just a few in tens of millions of collisions. To this end, the first critical task of a detector is to separate ordinary collisions from events, and this is the task of the *trigger*, which is placed in the so called counting room next to the detector. Raw data coming from the trigger are expected to have a rate of 200 Hz, with an event size of 1.6 MB, and are transferred to the CERN computing centre through dedicated high speed links. Here data are stored on tapes and on disk buffers for the first processing which produces *event summary data* (ESD). Event summary data have a size of approximately 500 KB.

The ATLAS Computing Model [87] describes the software used in the ATLAS experiment, the type and the flow of data coming from the detector and the deploy-

[3]The Higgs particle is a particle whose existence was predicted over 30 years ago, in order to explain the mechanism by which particles acquire mass.

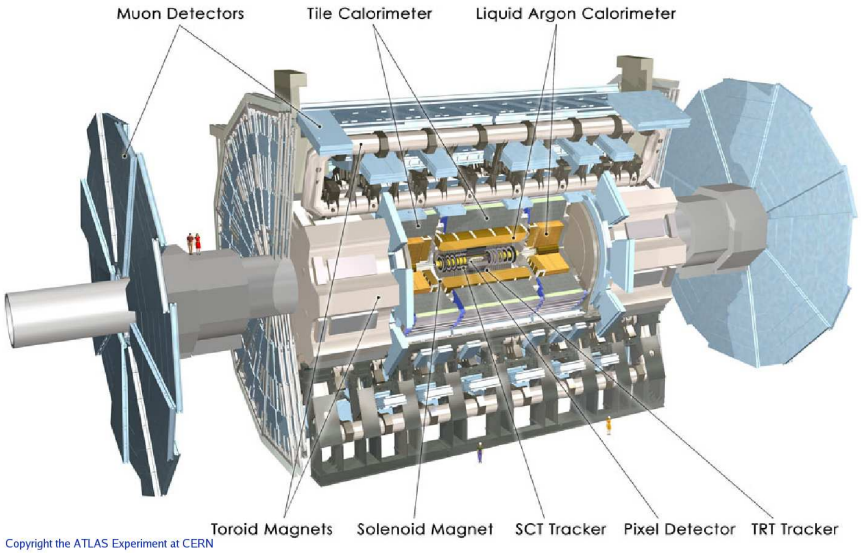

Figure 4.3: The ATLAS Detector

ment of the hardware infrastructure. Computations in ATLAS make wide use of Grid computing. Three Grid infrastructures, EGEE [7], OSG [31] and NorduGrid [27] are used.

The organisation and the flow of data and computations in the LHC experiments follow the MONARC model (Models of Networked Analysis at Regional Centres for LHC Experiments [25]). According to this model, sites participating in the ATLAS experiment have a hierarchical or tiered structure: CERN represent the Tier-0 site, where the raw data coming from the detector are stored and preliminarily processed. Raw data and the results of the first computations are then shipped to Tier-1 centres; in ATLAS there are ten world-wide distributed Tier-1 sites. Each Tier-1 site stores part of the raw data, performs further processing (or reprocessing), stores all the different versions of processed data and makes them available for Physics analysis. Each Tier-1 site is then responsible for the management of tens of Tier-2 sites where further analysis will happen. Tier-2 sites will also store part of Tier-1 data and provide simulation and analysis facilities; some of them will also perform some calibration operations[4] based on processing raw data. Additional resources will be available for

[4]What calibration means will be clarified in the rest of the section, where non-event data are introduced.

data analysis at Tier-3 sites. All the computations made on ATLAS data are done using a special software framework called Athena, which is a joint effort between the LHCb and the ATLAS experiment.

Like in all the other LHC experiments, the real challenge of the ATLAS experiment is to store enormous quantity of data, around 10 PB (that is 10^{15} bytes), per year, and make them available for scientific analysis spread over four continents. In this challenge we can find the real nature, and the innovative idea of Grid computing and Data Grids, explained in detail in Chapter 5.

4.2.2 Conditions Data

Data generated by the detector during a collision, are called *event data*. These data are mostly read-only. In order to perform event data analyses, another type of data is needed besides event data; they are called conditions, or non-event data and are usually modifiable. Conditions data are data produced during the operation of the detector and are required to perform reconstruction[5] and analysis. They come from the Detector Control System (DCS), from calibration and alignment elaborations and from other monitoring and book-keeping systems within the detector and the hardware and software infrastructure connected to it at the detector site. Calibration and alignment processing refers to the processes that generate non-event data that are needed for the reconstruction of ATLAS event data.

Conditions data vary with time, and are usually characterised by an *Interval Of Validity* (IOV), that is a time interval during which certain data have a fixed valued. Conditions data can have different payload structures. The payload is the part of the object where actual values reside. These values can be temperatures, voltages (measured and nominal), pressures and so on. Sometimes a condition object value can also be a reference to an external file where the actual value is stored. A condition object can also exist in different versions as a result of different computations performed during alignment and calibration on the raw data.

Conditions data are stored in relational databases. A MySQL database was tested in 2004, but the final solution will be based on COOL (see Section 4.2.3) and exploit Oracle Database technology features. Conditions databases are fundamental for the correct implementation of the ATLAS computing work flow, and the reliability and the performance of conditions databases are vital to the success of the experiment. This is why conditions databases will be replicated to Tier-1 and Tier-2 sites.

[5] Reconstruction is a specific kind of data elaboration to derive from the stored raw data the relatively few particle parameters necessary for Physics analysis.

4.2.3 The COOL API

The LCG Condition Database project (COOL) [4] was launched in 2003 with the goal of implementing a common persistency solution for the storage and management of the conditions data of the LHC experiments at CERN. It defines a common C++ API for conditions data storage and retrieval on different relational databases technologies (Oracle, MySQL, SQLite). It also support the use of FroNTier/Squid, a cache mechanism, that is explained in Section 4.4.2. At the moment COOL is used by Atlas and LHCb.

In the previous section we described the nature of conditions data. We saw that they are characterised by an IOV and a payload and that they can exist in different versions. Since there can be different sources producing conditions data, a condition object can be associated to a specific *channel*. Thus, usually, a set of measurements for the same condition object can be saved in a relational table with the following structure:

cond_data = (IOV, tag_name, channel_Id, payload)

The column "tag_name" is a tag associated with a particular version and the "channel_Id" is an identifier of the source channel. The payload can be a set of columns or just a column with an external reference.

Storing the payload in the row with its metadata is appropriate when the size of the payload is not too big and the payload has no meaning outside the context of an IOV. Storing the payload in an auxiliary table in the same database, with a foreign key referencing the IOV table, is appropriate when the data is naturally represented as a relational table but it is not always associated with an IOV, or it is shared by many IOVs.

4.2.3.1 Folderset arrangement

From the user point of view, the Condition Database (CondDB) looks like a tree (or a Unix filesystem), where the leaf nodes can hold conditions data objects. Leaf nodes are called "Folders" (equivalent to Unix files), while nodes that contain other nodes are called "FolderSets" (equivalent to Unix directories). The hierarchical structure allows a logical organisation of the conditions data; for example, one can put all the Folders for the conditions needed by a specific part of the detector in a dedicated FolderSet called *SubdetectorA*, or all the temperatures measured within the detector can go in a FolderSet called *Temperatures*. Thus, the user can create and manage Folders and FolderSets just like files and directories in a Unix filesystem.

IOV	Version Tag	Channel Id	Payload		
t1 to t2	-	1	10	10	comment
t2 to t3	-	1	13	10	comment
t3 to t4	-	1	11.5	10	comment
t1 to t4	-	2	42	6	comment

Table 4.1: Example of single version table

The COOL API provides two types of Folders: single-version and multi-version. The first one can only store objects with IOVs that do not overlap; given a time value, only one object exists and can be retrieved from a single-version Folder. In multi-version Folders, IOVs of different objects can overlap, and the user can use the version tag to discriminate between two or more objects in a specified time interval or time instant. In a multi-version Folder the most recent version of all the stored conditions objects is called the "Head" version, just like in a CVS repository. At a certain point in time, through the COOL API, it is possible to tag the Head version giving it a specific tag name. Users can then use this tag name to retrieve the specific version of an object.

It is worth noting that a Folder, which we may think of as a logical table in the Conditions Database, holds a set of measurements pertaining to the same condition data object. It also useful to note a possible usage of the channel_Id column. In case we have more data sources with the same payload structure, we could store data of these two sources in the same Folder and use the channel_Id to select one or the other source.

In Table 4.1 an example of a single version folder is shown with a table representation. We could think of this table as the one used to store temperature of a given "component X" of the detector, with 2 channels, expressed as an average value on a given number of samples. The table does not correspond to the real database table used by COOL to store the Folder but it is just a logical table that is useful to clarify the concept just explained. Being the table the representation of a single version folder, the version tag is not used, and rows (measurements of temperatures of component X) with the same channel_Id have non overlapping IOVs. For these rows, the payload is just two integer numbers (the average temperature and the number of samples) and a string (a comment). The first channel has 3 values from $t1$ to $t4$, while data coming from the second channel is more stable and has one value valid from $t1$ to $t4$.

In Table 4.2 an example of a multi-version folder for an analogous scenario is

IOV	Version Tag	Channel Id	Payload		
t1 to t2	1	1	12	10	comment
t2 to t3	1	1	13	10	comment
t2 to t3	2	1	11.5	100	comment
t3 to t4	1	1	14	10	comment
t3 to t4	2	1	14.8	100	comment

Table 4.2: Example of multi-version table

shown. The table shows a possible use of the version tag: for the IOV $[t1,t2)$ we have only one version, 12, which is the average value on 10 measurements. For the other two IOVs, $[t2,t3)$ and $[t3,t4)$, we have to versions: the first is the average value of the temperature with 10 samples, and the second version is the average value computed with 100 samples.

4.2.3.2 Basic COOL capabilities

This section collects the basic capabilities of COOL as they are explained in the Examples of the Doxygen documentation [3].

SingleVersion Storage A user can create a single-version Folder with a specific payload structure (payload specification) and insert/retrieve objects into/from it. The insertion can be done either with or without the bulk option. The bulk option allows storing a given number of objects in a single database transaction. The bulk size can vary.

MultiVersion Storage A user can create a multi-version Folder with a specific payload structure and insert/retrieve object into/from it. A user can also tag the Head version and later retrieve objects with a specific tag name.

Use FolderSets A user can create FolderSets just like he creates directories in a Unix filesystem. He can then create Folders inside FolderSets. He can also get specific FolderSets/Folders or list the Folderets/Folders inside the DB or inside a FolderSets.

Use Channels A user can insert/retrieve objects into/from a Folder by specifying a channel_Id.

Retrieve multi-channel objects in bulk mode A user can retrieve objects from a Folder in a given channel range.

UseTags A user can create tags (i.e., tag the current Head), remove tags, list current tags and print the content of a tag (i.e., retrieving objects with a given tag name).

ReTag A user can retag some objects removing the old tag and using the tagHeadAsOfDate call.

UseClob A user can set a field of the payload as CLOB (very long string).

4.3 Data Storage and Distribution

Two complementary storage approaches are used in ATLAS as well as in the other LHC experiments: file based and relational databases. Event data and large volumes of conditions data are stored in files. Relational databases are used when serialisation of data, transactional consistency and query based retrieval is required. Databases are also the choice when the access pattern is compatible with a centralised writer and distributed reader approach.

In both cases (files and databases), the ATLAS software environment is primarily object-oriented; the Athena framework, for example, is implemented in C++. File based storage of C++ object is done, in ATLAS, using ROOT I/O [80] and the POOL persistency framework [71]. For SQL based relational storage Oracle is the primary choice at CERN. However, MySQL has emerged as the preferred engine for installation at small research institutes. Vendor neutrality in the DB access, that is the possibility to use a standard interface to access databases of different vendors, has been addressed by the CORAL project [5]. Replication of Oracle databases at CERN to other technologies for distributed read-only use will be the key to allow distributed analysis of ATLAS data. The goal is to ensure that all ATLAS physicists, wherever they are located, will have access to all data of their interest.

For conditions databases, the ATLAS setup is shown in Figure 4.4. Data coming from the detector (on-line trigger farm) are firstly stored in the on-line database, which is an Oracle database with RAC technology [33]. The on-line database is the one where most of the data insertion operations (write accesses) happen. From the on-line database data are shipped to the off-line DB, another RAC installation, using a first level of Oracle Streams replication (explained in Section 3.5 and Section 4.4). The on-line database will only contain data valid in a short period, while the off-line

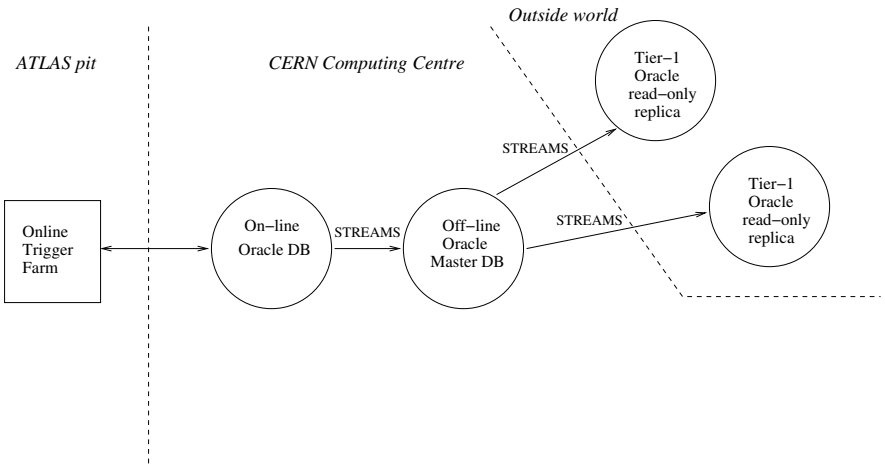

Figure 4.4: Conditions databases deployment in ATLAS

database will store all the data. The off-line database then acts as a master for the replication towards Tier-1 sites. Also this second level of replication is performed using Oracle Streams. The estimated yearly data volume for conditions data stored in relational databases is around 1TB.

4.4 The LCG-3D project

The LCG-3D project [57] (Distributed Deployment of Databases) focuses on the deployment of databases for the WLCG infrastructure. The CERN IT/DM (Data Management) group and LHC experiments are the main actors. The project, besides the coordination of database deployment and maintenance at Tier-0 and Tier-1 sites, is also investigating data distribution techniques for relational databases.

Within the 3D project, Oracle Real Application Cluster, or RAC [33] is the suggested technology to deploy high availability services. RAC technology provides a way for building database clusters with higher availability than single node databases and at the same time allows scaling the server CPU with the application demands. High availability is achieved through a software management layer that redirects incoming client connections to other nodes in case one of them is unavailable. By adding nodes to a cluster, the number of queries and concurrent sessions can be increased without affecting running services on the existing nodes.

In terms of data distribution, different techniques are available, each providing

different levels of support for fault tolerance, scalability and data consistency. For smaller data volumes and in reliable network environments direct synchronous data copying can be done for example through materialised views (see Section 3.1.1). In more complicated scenarios, with high data volume, unreliable links and frequent updates, Oracle Streams technology is expected to be the most reliable solution. Oracle Streams are discussed in Section 3.5. The next section reviews its application in the context of the LCG-3D project.

4.4.1 Oracle Streams for Tier-0 to Tier-1 replication

The Oracle Streams product allows the connection of single tables or complete schemas in different databases and ways to keep them synchronised. In the LCG-3D project, CERN is the master site and holds the tables that can be modified by users. Logs of these changes are automatically shipped to Tier-1 sites, where they are reapplied in correct transactional order. In case a database at one of the Tier-1 sites is not reachable (e.g. because of a network outage or database service intervention) logs (Logical Change Records, see Section 3.5) are kept on the master database system and are automatically applied once the connection has been re-established.

The database process which is capturing and queuing changes can optionally be executed on a separate machine to reduce the impact of the Capture process on the source database. Streams can be set up to provide a uni-directional or bi-directional replication between their endpoints. Even though bi-directional streams have been tested successfully, they add significant complexity to the deployment as conflicts between updates on both streams endpoints may arise and need to be properly handled. In the context of the LCG-3D project, hence, the single-master approach is the one that is currently in production. In Figure 4.5 an overview of the Streams setup managed by LCG-3D is shown.

The Oracle Streams technology has the advantage of being transparent to the database applications, which means that applications developed to work in a non-replicated environment will continue to work also with a Stream replication environment. Of course, to exploit the full capabilities of the replication, a middleware software that redirect users write/read accesses to different replicas must be present.

Many key WLCG database applications have been validated in the Streams environment and continue to function without any application change or application specific replication procedures.

Ten Tier-1 sites participate in the setup of the Streams environment within the LCG-3D project:

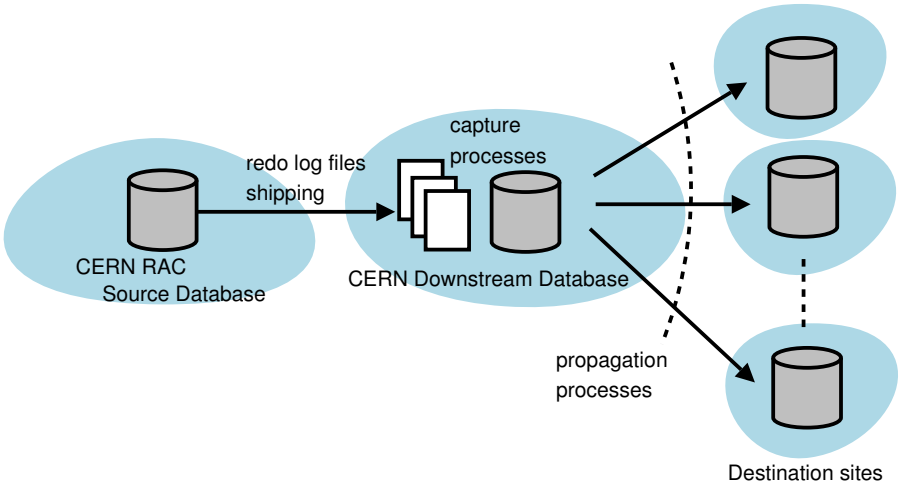

Figure 4.5: Streams Setup for the LCG-3D project

- ASGCC (Academia Sinica Grid Computing Centre, Taiwan)
- BNL (Brookhaven National Laboratory, USA)
- CNAF (Italian National Center for Research and Development about Information and Data transmission Technologies, Italy)
- GridKA (Grid Computing Centre Karlsruhe, Germany)
- IN2P3 (Institut National de Physique Nucléaire et de Physique des Particules, France)
- RAL (Rutherford Appleton Laboratory, UK)
- PIC (Port d'Informació Científica, Spain)
- NIKHEF (National institute for subatomic Physics, Netherlands)
- NDGF (The Nordic Data Grid Facility, Denmark)
- TRIUMF (National Laboratory for Particle and Nuclear Physics, Canada)

4.4.2 Using FroNTier/Squid for distributed caching

The LCG-3D project is evaluating FronNTier [61] as a data distribution technique for Tier-1 to Tier-2 replication. Distributed caching using FronNTier/Squid has been designed and implemented within the CDF (the Collider Detector at Fermilab) experiment. Its purpose is to deliver read-only data stored in a database to multiple users distributed world-wide. The cache system allows users to retrieve data even when they are decoupled from the central database. The generic architecture of a FroNTier/Squid installation is shown in Figure 4.6.

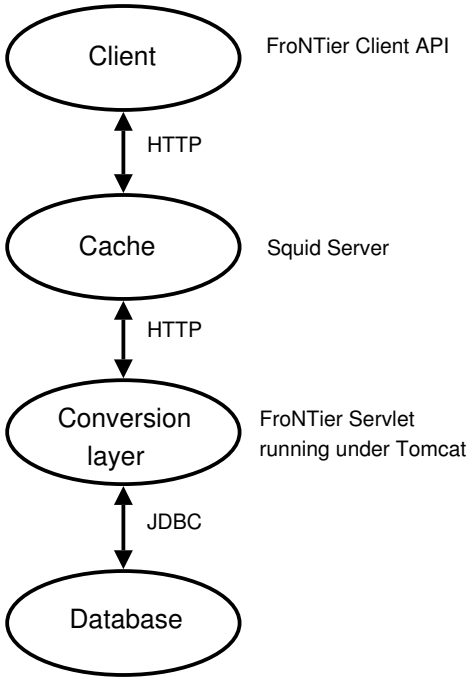

Figure 4.6: The FroNTier/Squid architecture

This architecture provides no single point of failure, scales easily to thousands of clients, and it significantly decreases the load on the central database.

Client queries to the database are translated by the FronNTier Client API library into HTTP requests that are delivered to a Squid web cache. Only in case the data of interest are not found in the cache, the query is sent to the database. Results of the query are then put into XML files by the FroNTier server and stored into the Squid cache so that the same query, the next time it is issued, can find the data in the cache.

This scenario works well when clients use repeatedly the same queries on the same read-only database tables, which is often the case in high energy Physics applications.

4.4.2.1 Consistency issues in FroNTier/Squid

One of the drawbacks of this solution is that the consistency among the central database and the distributed caches is not enforced. A squid cache server is not aware of possible database updates, so that applications running in a FroNTier/Squid environment need to be carefully designed and tested to avoid possibly subtle consistency issues caused by stale cached data. Periodically or on demand, cached objects must be refreshed.

The CMS experiment, which is one of the main user of the FroNTier/Squid solution, has agreed to a policy of never changing objects that are stored into the central database, and ultimately other cache refresh options will be implemented [50]. A mechanism that provides periodic cache refresh is implemented as an expiration time included as a meta tag in the cached XML file, which causes the file to expire the next day. This is an adequate solution for the short term. However, periodically reloading every cached object at every participating site will have significant performance implications.

4.4.3 Applications managed by the LCG-3D replicated environment

As we have already seen in the previous sections, data in the LHC experiments are mainly event and non-event data. However, the set of data used either directly or indirectly by users and services includes also different types of metadata. Several middleware services, for example the Replica Catalogue, use relational databases to store data. These databases need also to be replicated in order to obtain the advantages already explained in Chapter 3. In this section we briefly review the applications whose multi-tier replication is based on the LCG-3D replicated environment.

ATLAS Conditions Databases Tier-0 on-line to off-line and Tier-0 off-line to Tier-1 sites replication of ATLAS conditions databases is achieved with two Oracle Streams deployments (see Section 4.3). Data in conditions databases are inserted and retrieved using the COOL API.

AMI The Atlas Metadata Interface (AMI) is an ATLAS middleware service for dataset selection. AMI stores metadata about logical Physics datasets allowing users to choose the data they are interested in. AMI provides capabilities that

are complementary to the ones provided by DonQuijote-2 (DQ2), the ATLAS Data Management framework. AMI can be deployed using Oracle and MySQL databases.

CMS Conditions Databases In CMS, Oracle Streams is used to replicate the conditions database from the on-line system to the Tier-0 off-line system. Two levels of FroNTier/Squid caches are then used to provide read-only facilities at Tier-1 and Tier-2 sites.

LHCb Conditions Databases The LHCb experiment uses COOL to store and manage conditions data. Tier-0 to Tier-1 replication is done using Oracle Streams.

AMGA AMGA is a metadata catalogue used to store metadata associated to logical file names. AMGA is used in the WLCG project and the replication is done between Tier-0 and Tier-1 sites.

LFC LFC is the LCG File Catalogue (replica catalogue) used in the WLCG project. Its main purpose is to store the association between logical and physical file names (see Section 5.2.1.4). This catalogue is the results of a re-engineering process done on a previous catalogue developed within the European DataGrid project, that had serious performance issues.

Chapter 5

Grid Computing

Driven by the need of more computational power and storage capacity to solve complex data analyses, Grid computing is becoming a widely used computing model in many research fields. As an example, in Chapter 4 we presented the scientific challenge that physicists are facing at CERN, where the LHC will produce unprecedented amounts of data to analyse in order to investigate on high energy Physics. A Grid infrastructure will be used to store and analyse data coming from the LHC.

In this chapter we better define the characteristics of a Grid and we introduce the software used to manage a Grid, what we call *middleware*. We focus on the aspects that make the Grid original, even if, as we will see, it would be more appropriate to talk about an evolution more than a revolution in distributed and parallel computing. In many cases, in fact, well known standards and software are used to create services that are part of the middleware. We discuss more in detail data management and replication issues, where we also highlight the lack of a synchronisation service for replicated data.

The structure of this chapter is the following. In Section 5.1 we introduce the main concepts that help to define Grid computing and differentiate it from other methods of distributed and high performance computing. To give a practical example of Grid Computing we introduce the EGEE project, that manages the main European Grid infrastructure for e-Science applications. We also give an idea of what research communities are using the EGEE Grid. In Section 5.2 we explain what a middleware is and what are the main services that it uses to manage the resources and to execute user tasks. Two middleware examples are reviewed, the gLite middleware used in EGEE, and the Globus Toolkit which is widely used in many Grid infrastructures. We put more emphasis on data management topics, especially on services offered for

data replication, and we investigate on the support provided for replica consistency. In Section 5.3 we talk about Service Oriented Architectures and the Open Grid Service Architecture, which defines a model for creating and connecting Grid Services used in the Globus Toolkit starting from version 3. Tools for Data Access and Integration are introduced. In Section 5.4 we draw some conclusions about replica synchronisation in Grid computing, and state the main reasons for which we believe it is important to do research in this topic. The main issues in developing a Grid Replica Consistency Service are analysed, and the few previous and current efforts in this sector are cited.

5.1 Grids as a model for collaborative computations

Many different definitions have been used so far to define Grids and Grid Computing [67]. For our purposes, a Grid can be thought of as a distributed collection of heterogeneous computing, storage and network resources, that are bound together by a set of software services normally called *middleware*, and that are used by dynamic sets of people and institutions. The term Grid was introduced in [69], in 1998, with the idea of comparing this new distributed computing infrastructure to the electric power grid. The term Data Grid is often used to stress the properties of a Grid whose main purpose is the management of huge amounts of data[1], while the term Computational Grid is used when we want to focus on the processing power of a Grid. However, these are minor terminology aspects, and recently the two terms have started to be used together to define Grids as a Computational Data Grids.

As we are now able to plug a home appliance in the electricity socket in the wall, according to the "Grid vision" one day we could attach our personal computer to a Grid system to receive not electric power but computing and storage capacity to run our jobs. Of course this is a figurative way to give an immediate idea; the situation now is quite far from this vision, and Grids are mainly used in research fields where complex computations must be performed on large amounts of world-wide distributed data. The resources owned by a single institution are not enough to perform some tasks, this is why multiple institutions need to cooperate, sharing their resources, in order to achieve better results. If with the introduction of the World Wide Web he have been able to access and share world-wide distributed information, with the Grid we will be able to do the same with computational power and storage capacity.

[1] This is the case of the EU DataGrid project that was created for collecting and analysing data coming from the LHC experiments at CERN, that we discussed in Chapter 4.

Grid computing, at first sight, can look like another way to identify cluster computing or cycle-scavenging applications, but there are important differences to consider. Hence, let us refine the definition of Grid having a more detailed look into the main characteristics of a Grid.

When we say distributed collection of resources, we mean world-wide distributed; Grids today involve resources placed in different continents. The biggest difference that discriminates Grid computing from cluster computing is this one, and also the control of the resources. A Grid spans different administrative domains, there is no central point of management on the resources. A site that wants to join a Grid can share its resources still enforcing its access policies and keeping the ownership of them. This decentralised control over the resources is one of the biggest challenges of Grid computing. The number of resources used in Grid computing poses new challenges also in terms of scalability: thousands of users, and thousands of processors and disks connected through unreliable links must be managed in an efficient way.

We also mentioned that the resources are used by dynamic sets of organisations. In Grid computing we generally refer to Virtual Organisations (VOs) to identify a group of people that share a common goal that can be achieved using the Grid. For example, in the LHC Computing Grid [41] each LHC experiment (see Section 4.1) has its own VO, with many institutions participating in it. Virtual Organisations are dynamic in that people and resources forming a VO come and go; an institution or a single person can join a VO for a limited time as well as resources can be made available for some time, for special purposes, and then taken back. This dynamic property is another complicating factor in the development of Grid software.

Heterogeneity in the collection of resources being shared is another important factor. Jobs submitted to the Grid can be executed using different hardware and software components, and can read and store data from different storage technologies. Matchmaking services are generally used to find, dynamically, the most proper resources to execute a job; a job can have its constraints, like a hardware platform or the presence of some software programs. The user of the Grid is generally unaware of where his job will run, or where his job will read the data from. A big effort in developing Grid services is done to guarantee this access transparency.

Another important property of Grid computing is that of being based on open standards, and using as much as possible well-known and reliable protocols. Many services like file transfers and security services in fact rely on protocols that are not new in computer science; Grid tools are expected to use such protocols, enhancing them with new capabilities or adapting them to the new constraints posed in Grid environments.

5.1.1 The EGEE Project

As an example of an international Grid project, we now introduce the EGEE project, Enabling Grids for E-sciencE [7]. Funded by the European Commission, the project, in its first phase, started in 2004 with the goal of providing scientists with a production quality Grid infrastructure supporting applications from various scientific domains, like Earth Sciences, High Energy Physics, Biomedicine and Astrophysics.

The EGEE project ended in 2006 when the second phase of the project, EGEE-II, started. The EGEE-II project extends and consolidates the EGEE Grid infrastructure, which is shared with the Worldwide LHC Computing Grid Project (WLCG) [41]. The capacity provided by Grids like the EGEE Grid is much bigger than the typical capabilities of local clusters at individual centres. With EGEE multiple institutions can effectively collaborate towards a unique and sustainable tool for computationally intensive science (*e-Science*).

Today, the project brings together scientists and engineers from more than 240 sites in 48 countries world-wide. More than 5000 people are registered to use the infrastructure, but the number of people benefiting from it is more than twice larger. The set of resources shared in the EGEE Grid infrastructure add up to something like 30,000 CPUs and 20 PB of data storage. These numbers are expected to increase in the next few years.

Research institutions are not the only ones to benefit from the EGEE project, that is also highly committed to building strong relations with industry. Regular events are organised by the project to promote the adoption of Grid in industry, analysing business needs, highlighting technical and non-technical barriers and suggesting ways to overcome them.

EGEE is not only hardware infrastructure and middleware, these are just two of 11 activities within the project that also comprises things like training, support, work on standards, international collaborations and on relation with business communities.

5.1.1.1 Applications

High Energy Physics (HEP) and Biomedicine were the first two scientific groups that joined the EGEE project. More and more applications are starting to use the EGEE Grid infrastructure and today we have applications in Earth Sciences, Bioinformatics, Astrophysics, Multimedia and Finance, Astrophysics, Archaeology, Computational Chemistry and Geology. Researchers in those fields form Virtual Organisations, and collaborate, share resources, and access common datasets to solve computational and data intensive tasks. The original community that promoted the development of the

EGEE Grid infrastructure was the one working at CERN with the four LHC experiments (see Chapter 4). Other international HEP experiments are also making use of the EGEE infrastructure, including BaBar (B and B-bar experiment) and CDF (Collider Detector at Fermilab).

In Computational Chemistry, the initial and primary user is the GEMS a-priori molecular simulator [58]. Several applications have already been ported to the Grid to calculate chemical reactions, simulate the molecular dynamics of complex systems, and calculate the electronic structure of molecules, molecular aggregates, liquids and solids.

In Astrophysics several communities share problems of computation involving large-scale data acquisition, simulation, data storage, and data retrieval. In 2008 the European Space Agency (ESA) is expected to launch the Planck satellite with the goal of mapping microwave sky with an unprecedented combination of sky and frequency coverage, accuracy, stability and sensitivity. PlanckEGEE [36] is a project whose main goal is to verify the possibility of using a Grid Technology to process Planck satellite data. Another example application is the MAGIC [24] application, that runs simulations needed to analyse the data of the MAGIC telescope (located in the Canary Islands) to study the origin and the properties of high-energy gamma rays.

In Earth Sciences, the EGEE project is contributing to efforts in a large range of topics related to the earth's atmosphere, ocean, crust and core, including applications in earthquake analysis.

The biomedical community is benefiting from the Grid by enabling remote collaboration on shared datasets as well as performing high throughput calculations. The applications involve medical imaging, bioinformatics and drug discovery, with many individual applications deployed or being ported to the EGEE infrastructure. Among the most interesting projects, we cite the WISDOM (Wide In Silico Docking On Malaria) project, that makes use of the Grid for developing new drugs for neglected and emerging diseases with a particular focus on malaria.

Multimedia and Finance domains have just started to evaluate and use the Grid with EGEE.

To summarise, in the research environment, more and more application domains have started considering the use of Grid technologies as a good opportunity to improve their achievements. More computing power means having the opportunity to use more and more complex data analysis algorithms. Access to world-wide distributed storage facilities means increasing the scope of your analysis. Last but not least, the collaboration concept which is the most original contribution of Grid technologies offers to research institutes of medium-small size the possibility to join com-

plex and international projects which were before out of their scope.

In the commercial world, Grid technologies are mainly used in the finance sector to perform Montecarlo simulations and complex statistical analysis. In this sector, however, one of the key ideas behind Grid computing is missing: collaboration, that is, information and resource sharing. The collaboration of dynamic groups based on open standards is in contrast with some requirements of the finance sector, like privacy and competition with other companies. This is the reason why, for the finance sector, the distinction among Grid computing, High Performance Computing (HPC) and Service Oriented Architecture (SOA) is often unclear.

5.2 What is a middleware and what is it made of

In a high level view of the Grid, three main types of node can be found: Computing Elements (CE), Storage Elements (CE) and User Interfaces (UI). The term CE is used to identify a set of machines where a job-manager and a batch system are installed. The CE then will execute jobs using Worker Nodes (WN). For the purpose of this work, we can consider a CE as a single node where jobs can be scheduled and executed. Storage Elements are storage systems, like disk pools and mass storage systems with a "Grid Interface", that is the support of some data access and transfer protocols used in a Grid. Grid data are stored in SEs, and usually replicated on several SEs. To interact with the Grid, users use software installed on some machines, called User Interfaces, where they can log in and submit jobs, copy and move data and so on.

What in Grid computing is generally called middleware is a collection of software services, libraries and command line clients used to manage Grid resources and to provide services to users. The middleware is the glue that connects together all the actors and resources involved in a Grid infrastructure. The purpose of the middleware is also to hide most of the complexity of the Grid environment to end users, and ideally present them the Grid as a single large virtual computer that they can use to execute their computation.

In principle, a middleware should provide basic services for:

- Submitting and controlling user jobs. A user, especially in HEP, interacts with the Grid by submitting jobs, specifying the program to execute and the input data needed by the program. An identification number is usually returned to the user, that can use it to monitor the execution of the job and retrieve its output.

- Finding the most appropriate resources to execute jobs. Many heterogeneous computing and storage resources are available in a Grid. The middleware must be able to find resources that satisfy special requirements of the jobs. Policies related to different VOs, and efficient use of the resources, impose that the choice of the resources to use for the execution of a job consider also some accounting criteria in order to ensure that the job have access to the resources that it is allowed to use.

- Managing data stores, data access, transfer and replication. Different types of storage resources are available in a Grid, from disk pools to mass storage systems. The access to these technologies must be transparent to the user, that must be shielded from technology specific details. Services for accessing data, copying data from one site to another, and for replicating data must be provided.

- Collecting and publishing the state of the resources. Resources in a Grid are dynamic and heterogeneous. Their characteristics and status must be continuously available to users and middleware services in order for them to be properly used.

- Enforcing security and VO-specific policies. Security rules must be applied in order to avoid malicious use of resources. The users of a Grid must be authenticated and authorised to use the resources. Special policies related to virtual organisations must also be applied.

In the next sections we see two examples of middleware: gLite, the middleware used in EGEE, and the Globus Toolkit. We provide more details about data management services, highlighting the services used for replicating data, and the support for replica consistency. We also talk about Service Oriented Architectures, that is the model where Grid deployment is moving towards.

5.2.1 The gLite middleware

The middleware used by the EGEE and WLCG projects is gLite [13]. gLite builds on top of existing services developed in previous Grid projects, especially in EDG [11]. Many of these services have been re-engineered, others have been adapted to new sets of requirements, others have been designed and built from scratch.

Four areas can be used to group the basic capabilities offered by the gLite middleware, and in general by any middleware: Security, Information Service, Workload

Management and Data Management. In the next sections we review the main concepts and solutions provided in each of these areas and then we explain the main steps involved in a typical Grid user operation, the submission of a job. More emphasis is put on Data Management topics, since this is the area where the Replica Consistency Service is located.

5.2.1.1 Security

Security plays a fundamental role in the activities of a Grid. Data and resource sharing needs to be organised and managed considering all the potential security threats, like the use of resources by unauthorised users or the failure of critical operations due to an integrity violation in some client-server communications.

In the EGEE Grid infrastructure, the Grid Security Infrastructure (GSI) [17] provided by the Globus Toolkit is used. The GSI is based on, and extends, the Public Key Infrastructure, where a private key is owned and kept secret by an actor (user or server in the Grid), and a corresponding public key is publicly available to all the other actors who want to communicate with him/it in a "secure" way.

In general a secure communication between two actors is a communication where:

- each actor is sure about the authenticity of his counterpart (*authenticity*),
- the messages exchanged by the two actors are private, meaning that an external user intercepting the messages is not able to understand them (*privacy*),
- the messages exchanged cannot be modified by an external actor trying to alter the meaning of the conversation (*integrity*),
- the two actors are explicitly authorised to take part in the communication, meaning that one has the right to contact the other (*authorisation*).

The GSS-API is an IETF standard defined in the RFC 2743 and 2744 which defines an API that can be used by generic applications to achieve authenticity, privacy, integrity, and delegation. Delegation is the way a server can execute operations on behalf of a client.

The Globus Toolkit provides a set of security services, implemented following the GSS-API standard, in the form of Java packages, C++ libraries and command line client tools. These services cover privacy, authentication through X.509 certificates [39] , different methods for authorisation, and delegation and single sign-on through proxy certificates.

Coming back to the gLite middleware, let us see what a Grid user has to do in order to use the Grid in terms of security. First of all a user needs to obtain an X.509 certificate from a Certification Authority (CA) recognised by EGEE, as described in [54]. The certificate obtained by the user is usually valid for one year, and allows the user to submit jobs to the Grid and use Grid services. But, before the user can use the EGEE Grid infrastructure, he has to belong to a Virtual Organisation, and adhere to its rules. Different VOs are available, usually bound to a specific project; the complete list of VOs operated in EGEE and the way to join a VO can be found in the EGEE User and Application portal [8]. The next step is to obtain an account on a User Interfaces (UI) node; this is a machine where UI tools are installed, and a user should usually refer to the system administrator of his home institution to ask for an account. At this point the user, logging in to the UI machine, is able to execute a command line tool to generate a proxy certificate. This proxy certificate has a shorter validity than the user's certificate, and it is used to delegate the user credentials to the services the user wants to work with. With a valid proxy certificate, the user is able to submit jobs, retrieve information, and execute basic data management operations.

5.2.1.2 Information Service

The Information Service is used to publish users and resources information and make them available to other users and resources. Through the Information Service the resources available on the Grid and their status can be queried in order to the decide where and how a given job can or should be executed. Resource discovery is not the only usage of the Information Service. Accounting information must also be kept in order to monitor the usage of resource by different groups of users.

In gLite, two Information Services are used, the Globus Monitoring and Discovery Service (MDS) [20] and the Relational Grid Monitoring Architecture (R-GMA) [37].

The MDS is used to publish resources information for resource discovery purposes. Both static (i.e., the type of a resource) and dynamic (i.e., the current status) information can be published and queried. The MDS has a hierarchical architecture with servers at resource, site, and central level. Each resource, CE or SE, publishes information in its Grid Resource Information Server (GRIS). At the site level then, the Site Grid Index Information Server (GIIS) collects information from different GRIS providing a general view of the resources available at that site and their status. At the top level of the hierarchy a Berkeley Database Information Index (BDDI) is used to provide an overall view of the resources available in the Grid. MDS is based on an open source implementation of LDAP (Lightweight Directory Access Protocol) [32],

a database optimised for reading operations.

R-GMA is an implementation of the Grid Monitoring Architecture (GMA) proposed by the Open Grid Forum (OGF) [30][2]. From the user point of view R-GMA is a standard relational database where information about the system can be stored and queried using a subset of SQL. It provides a more stable and flexible solution compared to MDS. Its internal architecture is based on three main components: Producers, Consumers and the Registry. Producers provide information and register themselves to the Registry. Consumers can request information using the Registry to locate Producers holding the information they are looking for. Then, they can contact Producers directly to execute the queries. R-GMA, given its flexibility, can be used to store user defined information, in general for accounting purposes, to track the usage of resources by Grid users.

5.2.1.3 Workload Management

The Workload Management System (WMS) is the set of components that manage the job submission and execution process. A user on a UI node can submit a job providing a specification file written in JDL (Job Description Language) [65] where properties of the job to be executed are specified, like the location of the executable and the value of its arguments. Using JDL, the user can also list a number of files that the job needs to access during its execution, put some requirements on the machine where the job will be executed (e.g, platform, operating system) and specify where the output of the file need to be stored.

The WMS is a very complex system because it has to take decisions based on the status of the Grid resources at the time a job is to be executed, hiding most of the complexity of the Grid to users. The most proper CE must be found, and files needed by the job must be placed close to the machine where the job will execute. In order to do this, the WMS uses services provided by the Information Service and by the Data Management Service. During the job execution the user can retrieve status information and, in case, cancel the job, using the job identification number received when he submitted the job.

Many advanced features are available besides the basic ones, like the automatic resubmission of jobs in case of errors, the possibility to submit collections of jobs with dependencies and to retrieve the output of jobs in real time.

[2]The former GGF, Global Grid Forum.

5.2.1.4 Data Management

Data Management Services in the gLite middleware use the file as the primary unit for data management. A file is usually replicated for performance and reliability reasons, but after the creation a replica cannot be modified not to break the consistency among the replicas.

Different names are used to identify a file in the Grid:

- *GUID*, Grid Unique IDentifier: it is used to identify unambiguously a file into the Grid. Its format is: `guid:<unique string>` where the unique string is made of a MAC address plus a timestamp.
- *LFN*, Logical File Name: this is a human readable string, chosen by users, that they will normally use to refer to a file irrespective of its location. The format is `lfn:<any string>`.
- *SURL* and *TURL*, Storage and Transport URL: they refer to a physical file, meaning to a specific replica stored in a specific SE. The SURL (also known as Physical File Name, PFN) includes the name of the host where the file is stored, plus the path to its current location.
  The format is either `sfn:<SE_hostname>/<path>` or `srm:<SE_hostname>/<path>` depending on the type of SE, with or without an SRM interface[3]. The TURL has also a protocol specification and it is used to retrieve or move the file. The format is `<protocol>://<SE_hostname>:port/<path>`.

Files are stored in Storage Elements, but the mapping between GUID, LFN and SURL is kept on a replica catalogue. The catalogue supported by gLite is the LFC File Catalogue, that we discuss later.

Different Storage Elements and access protocols are supported in gLite. As regards the Storage Elements, three types of storage elements are supported: Castor, dCache and LCG Disk pool manager. Castor is a mass storage system that uses a disk buffer as front end to a tape robot. Files are kept on tapes and transferred to the disk buffer before being accessed. dCache is a system of disk pools managed by a single server which presents a single virtual file system to the users. This system can be used both as front end for mass storage systems or as disk server. LCG Disk pool manager is a lightweight disk server solution for small sites.

All the three types of Storage Element provide a uniform interface to users, the *Storage Resource Manager* (SRM) interface, which allows users to do operations like

[3] SRM is a standard interface for SEs, that we discuss later in this section.

requesting files, moving them from tape to disk buffer in case of mass storage system and reserving space, in the same way on all the SEs.

As we said, the catalogue used in gLite to locate files is the LHC File Catalogue (LFC). It contains the mapping between GUIDs and LFNs and between GUIDs and SURLs (the TURL can be obtained dynamically from the SURL). This catalogue is implemented using Oracle as backend database, but MySQL is also supported. Using Oracle, the catalogue can be replicated to several sites, but only with a single master configuration, that is, only one replica can be updated and the others are read-only synchronised copies. This type of replication is done using Oracle Streams (see Section 3.5), as we saw in the LCG-3D project in Section 4.4. The LFC publishes its endpoint in the Information Service, so that it can be found and queried by other services, like the Workload Management System. The LFC provides a command line interface through which users can query the catalogue to retrieve the list of registered files. In the LFC, LFN entries are organised like a Unix filesystem: users can create directories, list their contents, change their access rights and so on (for the complete list of commands see [54]).

For replica management gLite offers different solutions. The most proper way to manage replicas is through a set of command line client tools called *lcg_utils*. They provide commands to move data into or out of the Grid, to create or delete replicas, to move replicas from one SE to another one, to list the replicas of a given LFN, to retrieve the GUID from an LFN or SURL and so on.

It is worth emphasising that files are considered read-only: if a user modifies directly a single replica, all the other replicas became stale without warning the users of the Grid. There is no replica consistency service in gLite.

When using the LFC commands a user has also to be aware that the operations he does on the catalogue do not involve the files, but just the catalogue entry. For example, when removing an LFN from the catalogue, all the replicated files associated to that LFN are not removed from their SEs, but they will not appear as part of the Grid. Thus, these replicas will be unavailable but will still take space in the SE.

Another gLite Data Management Tool is the *File Transfer Service* (FTS) [12], which can be used to schedule file transfers between a source and a destination SE in the Grid. The FTS service is based on GridFTP, the standard protocol for data transfers in Grid computing. In order to use the FTS command line clients a user has to have a proxy certificate[4], after which he can use the command to submit a file transfer job either specifying sources and destinations as arguments, or giving a file

[4]In this case the proxy is a long-term proxy, created using a special service called MyProxy [26] server.

where sources and destinations are listed. The submission commands return to the user an identification string that can be used to query the status of the job or to cancel it.

Lower level tools also exist to execute data management operations, but users are encouraged to use the high level services that we just presented, mainly for two reasons: they are easier to use and they protect against possible inconsistencies that can be generated when using the lower level tools.

5.2.2 Data Management in the Globus Toolkit

The Globus Toolkit [15] is an open source software framework used for building Grids. It is being developed by the Globus Alliance [14] in collaboration with many others institutions.

Services and libraries of the Globus Toolkit are present in most of the Grids deployed all over the world, as we saw in the case of the EGEE project, where GSI, GridFTP and the MDS are used. Starting from version 3, the components of the Globus Toolkit follow the OGSA standard (see Section 5.3) and are built as web services. Some of the components offer both WS version (based on web services) and a pre-WS one, without web services.

The Globus Toolkit provides solutions for Security, Information Service, Job Submission and Monitoring, and Data Management. As regards the Security services, the main concepts have already been explained in Section 5.2.1.1. As Information Service the Globus Toolkit provides the MDS (see Section 5.2.1.2), both in a WS and a pre-WS version. The pre-WS version is the one used in gLite, while the WS version, also called MDS4, is an enhanced version with an OGSA based implementation. For the job submission, execution and monitoring part, the Toolkit provides a set of services know as Execution Management, also in this case with two versions, WS and pre-WS. The Execution Management is based on the GRAM (Grid Resource Allocation and Management) server [16] that provides, at least from a user perspective, capabilities similar to the gLite Workload Management System.

5.2.2.1 Data Management Services

In this section the basic data management services provided by the Globus Toolkit are reviewed. In particular, we highlight the differences with gLite and see, in detail, how the replication services are organised, and whether they provide any support for replica consistency. Three main components are provided, for file transfer, data location, and a higher level Data Replication Service.

5.2.2.1.1 File Transfers Two main components are provided to move files: GridFTP and RFT (Reliable File Transfer Service). GridFTP is the basic protocol that we already mentioned when talking about the gLite middleware. Here we give some more details about it.

GridFTP is a protocol defined by Global Grid Forum Recommendation[5] GFD.020, RFC 959, RFC 2228, RFC 2389. The Globus implementation of GridFTP is used in most of the Grids deployed nowadays. It provides a secure, efficient and reliable way of transferring files from or into the Grid as well as between two Grid sites. A server, a command line client plus a set of libraries to build custom clients are provided within the Globus Toolkit[6].

The GridFTP protocol derives from the well known FTP protocol, the Internet file transfer protocol. GridFTP extends FTP by adding some new capabilities like the multistreamed transfer, and integrating the GSI. It was built to provide a uniform data transfer protocol to be used in several storage systems in use by the Grid community.

Third-party data transfer is also possible using GridFTP. It means that a file transfer between two machines can be initiated and controlled from a third machine. Parallel, or streamed data transfer improves the aggregate bandwidth over using a single TCP stream. Another bandwidth improvement comes from the striped data transfer, meaning the transfer from multiple sources to a single destination when data are partitioned among multiple servers. When files are large, it is sometimes desirable having the possibility to transfer just a "piece" of file. In GridFTP this is possible with the partial data transfer, where the transfer of a piece of a file starting at a particular offset can be done. These are the most interesting features of GridFTP; the complete set of commands or server options can be found in [19].

The Reliable File Transfer Service builds on top of GridFTP and provides a web service oriented interface to it, plus new features to recover from transfer failures, both on the client and server side. The RFT Service is an OGSA compliant Grid Service, and allow users to schedule one or more file transfers and monitor them asynchronously, as we already saw for the gLite File Transfer Service.

[5]The Global Grid Forum, now Open Grid Forum (OGF) [30] is a community of users, developers, and vendors whose mission is to accelerate the adoption of Grid technologies by providing an open forum where the main activity is to promote Grid software interoperability through the definition of open standards. Besides collecting and reviewing proposal for protocols, services and best practises in the use of Grid technologies, the OGF also organises events that bring together Grid experts from all over the world.

[6]GridFTP C++ libraries are used in CONStanza, as we explain in see Section 6.4.3.5.

5.2.2.1.2 Replica Location As we already saw in gLite, data can be replicated on several storage systems of a Grid environment. When multiple copies of a same data item exist, we need a way to locate them. We already saw in gLite the naming scheme (GUID, LFN, SURL and TURL) and the catalogue used to locate replicas, the LFC. In Globus, just LFN (Logical File Name) and PFN (Physical File Name) are used, and their association is kept in the Replica Location Service (RLS). In this case, the LFN plays the role of unique identifier, like the GUID in gLite.

The RLS is made of one or more Local Replica Catalogues (LRC) and zero or more Replica Location Indexes (RLI). An LRC holds the mapping between LFNs and PFNs. A client can query an LRC to find where the replicas associated with a given LFN are physically stored. An LRC can be deployed on MySQL, PostgreSQL, and Oracle. When multiple LRCs are present in the same Grid environment, a Replica Location Index can be used to store the LFN mappings present at each LRC: in this way, a client that wants to find the physical location of a replica, instead of contacting all the LRCs, can query an RLI to find which LRCs holds the mapping for a given LFN. Then it will contact directly that LRC to retrieve the PFNs associated to that LFN. Entries in the RLI are pushed in by the LRCs, which periodically must refresh these information that have a limited life time.

In a small environment, with up to ten sites, a fully connected deployment of LRCs and RLIs is possible. In this case an LRC is present at each site[7] together with an RLI. The LRC stores mappings at its own site, and the RLI stores the LFNs mappings of all the LRCs. Thus, an LRC has to publish its information in all the indexes. In this way, every site has a complete picture of the overall system, and a user can query a single index to find out in which LRC the mapping he is looking for is stored. An example of fully connected deployment with 3 sites is depicted in Figure 5.1.

Entries in the RLIs are the same and for this reason the figure shows only one case. In this example a client looking for replicas of the logical file LFNa, will first query an RLI. The RLI will reply saying that in order to find all the replicas it has to contact all the LRCs, because all of them hold at least one replica. Instead, to find replicas of LFNe for example, just LRC2 must be contacted.

In larger deployments, this configuration will not scale since every LRC has to periodically send refresh information to all the RLIs. In these cases the LRCs can be configured to publish their information only to some indexes. Thus, indexes will not have a complete view of the overall system but just a partial one, and one or more RLI could be contacted by clients in order to retrieve the needed information.

[7] We assume that a site has a single SE.

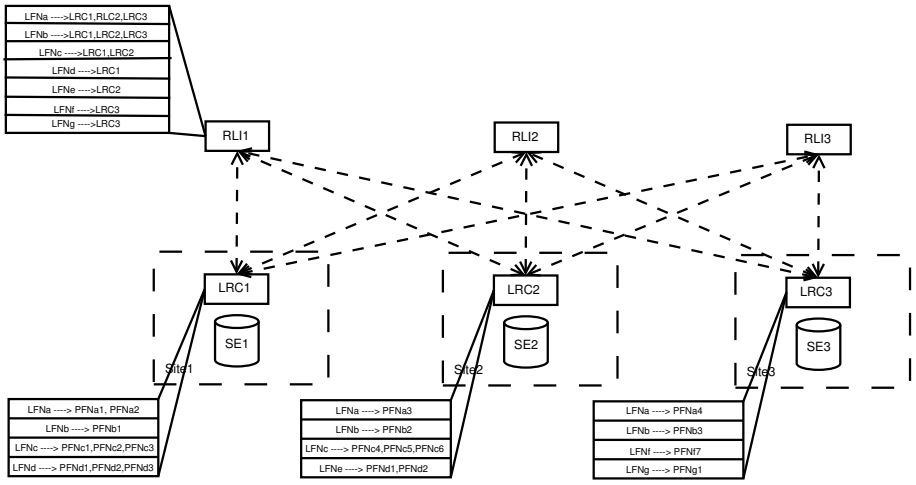

Figure 5.1: RLS, fully connected deployment

It is worth noting that the RLS does not perform any consistency check. When two replicas are registered into the LRC the system does not do anything to check whether the two replicas are actually exact copies. In the same way, if a replica is modified, the modifications are not propagated to all the other replicas so that what in the RLS appear to be copies of the same LFN in reality are different. No support for replica synchronisation is provided, and users have to be careful and manually remove replicas when executing some updates. This lack of replica synchronisation has been already found in the gLite Data Management Services, and is one of the main factors that led to the work done in this thesis.

5.2.2.1.3 Data Replication For what we have seen so far, the Globus Toolkit provide file transfer services and replica location services. This means that if a user wants to replicate data, and make the new replica visible into the Grid, he has to do a file transfer (copy the original file to a new destination) and then manually insert into the LRC an entry for the new file, specifying that it is a copy (replica) of an existing LFN. In version 4.0 (the current one) of the Globus Toolkit a Data Replication Service (DRS) is presented as a technical preview.

The DRS builds on top of the RFT Service and the RLS. The DRS is a web service compliant with the OGSA specifications and provides a command line client to create a replication task and control it. The task, that is the replicas to create, is specified in a request file that must be given as argument to the command

`globus-replication-create`. The user can then monitor and in case stop the replication task using others command line clients [18].

5.3 Service Oriented Architectures and the Grid

In the previous sections we introduced the basic services provided by the Globus Toolkit, and mentioned that WS and pre-WS versions exist for some services. Starting from version 3, the Globus Toolkit is based on the Open Grid Service Architecture (OGSA) [70], that merges concepts from Web Services and Grid computing. In OGSA a Grid architecture is seen as a Service Oriented Architecture (SOA) or more specifically as a collection of *Grid services* that can be dynamically created and connected to provide services. A Grid service, as defined in OGSA, is a web service that implements a specific interface and provides a standard way to manage the service lifetime. A standard approach to notification can also been implemented by a grid service, so that a user can subscribe to a service in order to receive messages about significant changes in the service internal state. Each grid service is then associated with a set of *service data*, through which a user or application can discover a grid service that provides the desired capabilities.

Grid services can be hosted in an application server environment such as J2EE or .NET, or deployed as self-standing processes. In complex environments, however, the services will span heterogeneous, geographically distributed environments. Exploiting the platform independent capabilities of web services, grid services can effectively support the operations performed by Virtual Organisations.

5.3.1 Integration of databases into the Grid with OGSA-DAI

While early Grid applications were only using file based data access, nowadays the need of databases and their integration with middleware services is widely recognised. Databases are not only used to store application data but also middleware service metadata. OGSA-DAI [29] is "a middleware product which supports the exposure of data resources, such as relational or XML databases, on to Grids". It is part of the Globus Toolkit and its main goal is to provide a service-based interface to different types of databases. At the moment OGSA-DAI supports MySQL, IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL and the eXist XML database. An important functionality of OGSA-DAI is the possibility to group several database queries into a single service request, which will be exploited by the Distributed Query Processing service described in the next section. Different options for delivering results

are also possible.

In OGSA-DAI the main role is played by *data services*, that are specific implementation of grid services used to manage the access to a data storage resource. Using OGSA-DAI a user can, with a single request to the OGSA-DAI data service, send an update to a database, then query this database and deliver the results of the query to a third party. The data service interface is based on *activities*. Each activity (a query to a database, an operation on the result or the delivery of the result) is mapped to a Java class that is responsible of its implementation. This architecture provides the way to extend OGSA-DAI with operations that are relevant to a specific application, for the purpose of data access, transformation and delivery tasks.

5.3.2 OGSA-DAI Distributed Query Processing

OGSA-DAI Distributed Query Processing (DQP) [62] is a service to perform distributed queries over a set of databases in a service oriented middleware as defined in OGSA. DQP itself is built as an OGSA-DAI service. It provides a way to execute parallel queries over a set OGSA-DAI and other Grid Services. OGSA-DQP's architecture includes a *coordinator service*, to interact with a user request. The coordinator receives a query plan as an XML document (*perform document*) and breaks it into multiple sub-plans that can be executed in parallel by *evaluation services* on the actual data resources. As we saw in OGSA-DAI, the results of the distributed query can be returned synchronously or asynchronously to the user or sent to another service. This allows users to combine data integration (query results from different data sources) with analysis capabilities (results sent as input to an analysis service).

5.4 Replica consistency in Grid computing

In Section 5.2, when we reviewed the gLite and the Globus Toolkit software, we saw that both middleware solutions provide services for replicating data and for locating replicas with replica catalogues. However, none of them provide any support for replica synchronisation. A user that wants to update a replicated file, can select one of this replica and modify it, but then he has to be aware that this would cause the other replicas to become stale. What a user can do is to manually remove all the other replicas, and create new replicas using the one that he previously modified. It is clear that this is not a good solution; it is complex, error prone, and cannot be used when updates are frequent. Moreover, when users want to update the same replica, they have to coordinate themselves in order to avoid inconsistencies

The lack of replica consistency support is partly due to the fact that many applications that are driving the development of Grid middleware expect to use modifiable datasets in the future, but currently use mostly read-only data. As a consequence, requirements for replica consistency are still unclear. More precise requirements will be defined when users begin to try new models of computation. Even if in most applications the read-only assumption is reasonable and allows for a simplification in the data management architecture, research in replica consistency issues is important [66].

Research in replication algorithms has shown that replica synchronisation comes at some cost in terms of data availability, so that only certain applications can fully profit from replicated data with update features. Just as replica consistency has become an essential property in distributed databases and filesystems, the same will obviously occur in Grid infrastructures. Further, considering that Grid computing is a rapidly emerging technology, it is likely that new applications, outside the scientific field, will arise in the next few years, providing more requirements for the implementation of a Replica Consistency Service.

5.4.1 Issues in designing a Replica Consistency Service

The design of a Replica Consistency Service as part of a Grid middleware must face many difficult issues that derive from specific properties of a Grid environment. In general, being replica consistency a highly application-specific problem, designing one consistency management mechanism for different applications requires finding trade-offs on many different design choices. In the next paragraphs we review some of the most important issues that need to be dealt with, providing hints for the design of a Replica Consistency Service.

5.4.1.1 Scalability

First of all, any Grid infrastructure involves the management of many sites, hence, in case of flat files, it is likely to have to deal with several thousands of replicas, some of which could not be continuously available. Thus, update propagation algorithms must be properly designed to provide good performance also with large numbers of replicas. Keeping the design simple can be the key to success; whenever possible, single master solutions are the recommended way to provide fast read access and high data availability.

5.4.1.2 Security

Security issues must be considered in the development of a RCS, like in any other middleware service. Communication with the service should be secure; this means that the service should deal with authorisation, authentication, privacy, and integrity issues. The Grid Security Infrastructure (GSI) provided by the Globus Toolkit is widely adopted as an integrated solution to security problems and it is based on the public key infrastructure. The GSI can be easily integrated in a Grid service.

5.4.1.3 Replica Location

Replica location services and replica catalogues are used in Grid middleware to store the association between a logical file and all its replicas. Among the most used implementation we cite the Globus Replica Location Service (RLS) and the LCG File Catalogue (LFC) seen in Section 5.2. The RCS has two options: interfacing with these catalogues or implementing its own replica catalogue. Both options have advantages and disadvantages. Using an external replica catalogue would avoid duplicating information and complicating the system. On the other hand, the integration with an external service should be carefully planned and would require such catalogues to be modified. For example, not all the logical files registered in a replica catalogue need the consistency management, like read-only files. For files that do require consistency management, some new attributes (e.g. master/slaves, fresh/stale, version number) should be added to each replica's metadata.

5.4.1.4 Efficient file transfer

An efficient file transfer tool should be used for update propagation. File transfer services for Grid computing are normally built on top of the GridFTP protocol. The RCS should use either GridFTP or higher level services to efficiently propagate updates to possibly thousand of replicas. Most of the Storage Elements support the GridFTP protocol, making it an ideal choice to solve the file transfer issue in the RCS.

5.4.1.5 SE heterogeneity

A Grid connects many different resources. Storage Elements, where files are stored, can have different implementations and different access protocols. Although a standard interface could be available in the next few years (SRM), a RCS should interface with different SEs. Lock management features should be provided by the SE since, in

certain scenarios, the access to a replica may need to be blocked to avoid concurrent access.

5.4.1.6 Disconnected nodes

The RCS should be able to perform the synchronisation of replicated datasets even when some of them are not available. Quorum mechanisms could be used to ensure that an operation is performed when at least a given number of replicas are available, and it should be possible to select this number depending on the application requirements. Basically, a quorum system should be used to deal with disconnected sites (see Section 3.2.1.1). Synchronisation of unavailable replicas should be retried as soon as they become available.

5.4.1.7 Metadata Consistency

The RCS should provide synchronisation capabilities both for applications and middleware services. Many middleware services in fact use replication for fault tolerance and reliability. One example can be found in the Globus RLS, where catalogues are replicated but consistency management is not supported. This led us to consider, as already stated in this work, the consistency of both files and databases. Although they present different characteristics, file synchronisation and database synchronisation have similarities that should be exploited to provide a general and flexible Replica Consistency Service.

5.4.2 Previous and current efforts in Grid replica consistency

The first work on replica consistency in Grid computing is dated August 2001 [56]. In this work the problem of files, objects and database consistency in a Data Grid is analysed. Use cases for High Energy Physics (HEP) applications are presented and different possible inconsistencies scenarios are introduced. Here the Consistency Service proposed is set on top of other Data Management middleware services like File Transfer, Replica Catalogue and Replica Manager. It is also stressed that the Replica Consistency problem is highly dependent on the application requirements and the data model. Thus, the envisioned Consistency Service must support different synchronisation protocols. It is also noted that consistency issues can derive not only from updates on existing data but also from creation of new data. No prototypes nor simulation studies were done in that work.

The first practical study about a possible synchronisation service has been done in [44], where the OptorSim Grid Simulator [53] was enhanced with a consistency module to simulate two replica synchronisation algorithms. A synchronous and an asynchronous protocol have been simulated on replicated files in a Grid made of 3 CE and 10 SE with job requesting logical files replicates on all the SE. The probability that the file access was a write operation has been used as a parameter to study how many times read and write access were done on the up-to-date replicas.

In [90] two coherence protocols for Grid applications are presented. In the first one, a lazy approach is used, where a replica is updated only before being accessed. A central catalogue is used to store metadata information, including the timestamp of the last modification of each replica. One replica acts as the *home* for all the others, meaning that it always has the most up-to-date information (it is a master replica). The update of a replica is done comparing the timestamp of the replica we want to access with the one of the home replica. In case the home replica is newer, a "diff" between the two replicas is done and the stale replica is synchronised. The second protocol uses an aggressive-copy approach, where the replicas are updated as soon as the home replica is modified, using a push method. The problem of temporarily disconnected replicas is not dealt with. No simulations nor prototype implementations of these two protocols are presented.

In [81] an architecture called Adaptable Replica Consistency Service (ARCS) for dealing with replica synchronisation in Grid computing is presented. Grid sites located closely are organised into a region, and the consistency of replicas in each region is managed by a *Region Server*. Each region has only one master replica, which is responsible of write operations and of synchronising the read-only replicas (slaves) in its site. The overall architecture however has a multi-master configuration, with one master located in each region. The access serialisation among the master replicas is done using a *key*, that is a token, kept is the replica catalogue, which grants a master replica the right to be updated. Thus, a synchronous approach among the master replicas is foreseen. A simulation model, implemented on top of the OptorSim Grid Simulator is presented.

For what concerns database replication in a Grid environment, in [49], a novel Grid Database Replication Model is presented to provide a framework for replicating and synchronising databases in a Grid environment. The architecture uses existing Grid services like the Metadata Registry (replica catalogue) and the Transfer Service (GridFTP), and relies on existing proprietary database replication solutions. To achieve heterogeneous database replication, the main idea is to provide a standard interface towards different database replication mechanisms, to perform operations like

log extraction (capture) and export (to create a replica) on the source database and log application (apply) and import on the destination database. Between the capture and the apply process, a Grid file transfer service is used to propagate data. Although it is planned to address both homogeneous and heterogeneous database replication, the current prototype implementation is based on IBM DB2 databases and DB2 SQL Replication [21].

The problem of concurrency control in distributed heterogeneous databases in a Grid environment is studied in [85]. This work focuses on the problem of consistency in distributed databases when data are partitioned among different databases in a Grid, and transactions must be properly scheduled when accessing data on multiple databases. Due to the lack of a global management system, it is shown how incorrect schedules can generate inconsistencies among the databases. A Grid Concurrency Control (GCC) protocol is proposed, extending the serializability concepts seen in Chapter 2 to heterogeneous databases. The discrete-event simulator CSIM [6] has been used to study the performance of the protocol.

Part II

System Details

Chapter 6

CONStanza, the Replica Consistency Service for Data Grids

In Chapter 5 we saw two important and representative examples of middleware, gLite and the Globus Toolkit, and analysed the support they provide for data replication. We also stated that no support for automatic synchronisation of replicated data is provided, leaving to the users the task of executing manual operations in order to solve potential inconsistencies. In particular, in gLite, file replicas are considered read-only, but users do have the possibility to modify a replicated file introducing inconsistencies. We explained the reasons why the replica consistency problem has not been faced so far, and we also stressed the reason why we think it is important to do research in this topic.

In this chapter, the main deliverable of this work, the CONStanza project, is presented. The CONStanza project started in 2003, as an investigation on possible algorithms for the consistency of replicated files in a Data Grid [44]. After that experience within the INFN-Grid project it was decided to start a real implementation of a Grid Replica Consistency Service.

This chapter presents the design and the implementation of such a service. In Section 6.1 we quickly review the domain of the system we want to design, recalling the main concepts and the terminology that we will use in the following sections. In Section 6.2 we state the requirements of the system, separating functional requirements from non-functional ones, and we present a use case model with a detailed specification for the main uses cases. In Section 6.3 we analyse the functional re-

quirements in order to formalise some concepts for a smooth approach to the design phase. We present the main analysis classes of the system, their associations and how they interact in order to realise the use cases. In Section 6.4 the architecture of the system is presented, and more details for each subsystem are provided. In particular, we show how CONStanza implements the synchronisation of heterogeneous databases, considering also non-functional requirements like fault-tolerance.

6.1 Domain analysis

A domain analysis has already been done when we talked about Grid computing, and specifically about the middleware and Data Management services, in Section 5.2. In this section we briefly review some of the most important concepts that constitute the domain of the system we want to develop, the Replica Consistency Service. In what follows we refer to a generic Grid environment where we distinguish three kinds of Grid nodes: *Computing Elements* (CE) provide computing power, *Storage Elements* (SE) provide storage capacity, and *User Interfaces* (UI) are the hosts that users connect to for accessing the Grid. Several services compose the Grid middleware, providing such capabilities as Grid-wide job scheduling, resource allocation, security and data management. The latter class of capabilities includes the *Replica Management Service* (RMS) and it is also where our Replica Consistency Service is located.

A *Virtual Organisation* (VO) is a group of people sharing computing resources and collaborating towards a common goal according to well-defined rules. This concept plays a vital role in the mechanisms that control access to Grid resources and in the management of the resources themselves.

The Replica Management Service is a middleware component that provides data management capabilities to Grid users. One example of RMS has been already presented in 5.2.1, where we talked about the `lcg_utils` command line clients provided by the gLite middleware.

As discussed in Section 3.1, data replication imposes a distinction between a logical dataset and its physical instances, i.e., its replicas. Replicating a dataset means that several physical copies of its content exist on different Grid nodes, i.e., on different SEs. In the case of file replication, a unique name (GUID) identifies the set of replicas for a file, and Logical File Names (LFNs) are mapped to the unique name. Logical file names can have some metadata associated to them. Logical file names are an abstraction mechanism, used to identify a group of replicas, but the GUID is the unique identifier that identifies unambiguously a set of replicas. File replicas

are stored in Storage Elements, of different types, with different access protocols. A replica on a SE is identified by a Physical File Name (PFN). Replicas of the same logical file can be independently modified, but the RMS does not provide any mechanisms to enforce consistency among them.

Besides Storage Elements, relational databases are also used to store data. Replica catalogues, like the RLS and the LFC discussed in Section 5.2 are deployed using Oracle, MySQL, Postgres and SQLite databases. Such replica catalogues can be replicated, but this replication is based on proprietary solutions (like Oracle Streams, explained in Section 3.5) and supports only homogeneous replication, i.e., among databases of the same vendor.

6.2 Requirements

Collecting and defining requirements is the first and most important phase of a software development process. Starting the design of a system without having clarified what the system should do is one of the main causes of failure in software projects. In this section we define the requirements of the Replica Consistency Service; we separate functional requirements (what the system should do) from non-functional requirements (properties or constraints of the system). Then we start building our use cases providing details for the most important ones.

6.2.1 Functional Requirements

Functional requirements describe *what* the system should do. It is important, in this phase, to avoid mentioning *how* the system should do something. Each requirement is identified with a label *Fi* and a priority level is associated to it; in some cases a note will be appended to add information about the requirement.

F1 The RCS shall allow clients to maintain consistency among replicated files. To this purpose, it must be able to propagate update information and apply them to all the replicas of a a logical file.

- *Priority:* High.
- *Notes:* This is the main requirement, for flat files. The synchronisation of files is triggered by a user request on all the replicas of a given logical file. A requirement for an automatic procedure to synchronise replicas with a given periodicity has a low priority.

F2 The RCS shall allow clients to maintain consistency among replicated heterogeneous relational databases.

- *Priority:* High.
- *Notes:* This is the main requirement, for relational databases. For databases, on demand synchronisation and automatic synchronisation, with a user-defined periodicity must be provided.

F3 Strict consistency is generally not needed, lazy algorithms must be preferred (see Section 3).

- *Priority:* High.
- *Notes:* The details about the algorithm that will be used for Update Propagation (UP) is not important in this phase. For flat files a synchronous approach will also be provided, but with a low priority.

F4 The RCS shall allow clients to specify a quorum value for an update operation.

- *Priority:* High.
- *Notes:* This is a requirement that deals with the possibility of having some replicas unavailable at the time an update operation is done. The quorum is the minimum number of available replicas required for an update operation.

F5 The RCS shall allow clients to retrieve the list of up-to-date replicas of a logical dataset and the update status of all the replicas of the dataset.

- *Priority:* High.
- *Notes:* A replica can be stale with respect to some other replicas of the same logical dataset when an update operation is running (not yet completed) or when such a replica is unreachable. In order to avoid stale reads the user must be able to retrieve update information about the replica that he wants to access.

F6 The RCS shall allow clients to check the status of an update operation.

- *Priority:* High.
- *Notes:* When a user requests an update operation, he must be able to monitor the update process.

F7 For flat files, the RCS shall work independently from the Replica Management Service (RMS).

- *Priority:* High.
- *Notes:* We will start designing the system to work independently from the RMS. The integration with a RMS will have a low priority.

F8 The RCS shall allow privileged users to register/unregister logical datasets as well as their replicas, putting them under or removing them from the RCS control.

- *Priority:* High.
- *Notes:* The RCS maintains consistency only of those replicas that have been previously registered to it.

F9 The RCS shall allow privileged users to set the UP protocol and its details, if any, but, at any time, there can only be one active protocol responsible for all the datasets and all the update operations (one-for-all).

- *Priority:* High.
- *Notes:* The consequences of a change of protocol when some update operations are still running must be considered.

F10 The UP protocol can be different for different types of dataset (one-per-dataset-type).

- *Priority:* Low.
- *Notes:* At a certain point in time the RCS can use a protocol to update flat files and a different one to update databases.

F11 The RCS shall allow different UP protocols to be used by different VOs (one-per-VO).

- *Priority:* Low.
- *Notes:* Assuming that the RCS is unique in a Grid architecture, and serves different VOs, it must be able to use different UP protocols for different VOs. The feasibility of using different UP protocols at the same time needs to be evaluated.

F12 The RCS shall allow clients (or privileged users) to set/change the UP protocol of a logical dataset: the protocol will act in the same way for all the update operations concerning a logical dataset (one-per-dataset).

- *Priority:* Low.

F13 The RCS shall allow clients to choose the UP protocol of any update operation (one-per-operation).

- *Priority:* Low.
- *Notes* This is a very complex case that needs to be refined.

F14 The RCS shall allow an external monitoring system to retrieve useful data about the consistency of replicated data.

- *Priority:* Low.
- *Notes:* We have to provide the system with proper methods that can be used by a monitoring system for retrieving status information about the service. The integration with a specific monitoring system must still be defined.

F15 The RCS shall allow privileged clients to choose an update mechanism.

- *Priority:* Low.
- *Notes* We will start with a fixed update mechanism, that is log based for databases and file replacement for flat files.

F16 For flat files, the RCS shall work with Replica Management Service (RMS).

- *Priority:* Low.
- *Notes:* We will start designing the system to work independently from the RMS. The integration with a RMS will have a low priority.

Requirements **F9** through **F13** define the flexibility/configurability of the protocol, specifying different numbers of degrees of freedom.

Requirement **F9** is the less flexible in that it provides a fixed protocol to deal with the consistency of all the datasets. It is likely that more flexibility will be required, for example, to use a different protocol for files and databases (Requirement **F10**). Requirement **F11** instead would allow different VOs to set their own protocol; this flexibility level could be useful in case VOs can be logically grouped basing on their

RCS usage pattern. Increasing again the degree of flexibility, Requirement **F12** could allow privileged users to choose the best protocol for each logical dataset; some service clients could need to use different protocols for different datasets, in case the requirements in terms of consistency is logically bound to the semantics of the data. Requirement **F13** is at the other extreme in terms of flexibility: in this case even the same file can be treated differently depending on some dynamic criteria like the user requesting the operation or the execution time of the operation. In Figure 6.1 these different approaches are ordered along a flexibility axis.

In case a protocol is bound to a VO, to a dataset or to a dataset type, it should be clarified whether this association can change over time. This case could present some issues when a protocol change overlaps some update operations involved with the change.

At the beginning, we decided to focus our attention on the simplest case, that is Requirement **F9**.

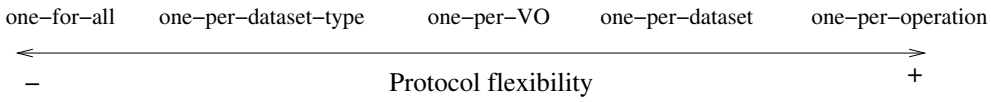

Figure 6.1: Flexibility provided by different consistency protocol setups

The functional requirements are summarised in Table 6.1.

6.2.2 Non-functional Requirements

Non-functional requirements cover general design guidelines and system constraints that can be stated during the definitions of the requirements.

NF1. The RCS shall use a strictly consistent internal catalogue.

- *Priority:* High.
- *Notes:* The RCS, having itself a catalogue, should make use of replication, thus of consistency mechanisms too. This can be done using existing solutions for homogeneous databases in a local area network, like Oracle or MySQL single-master replication.

NF2 The RCS shall cope with disconnected sites.

- *Priority:* High.

Priority	Requirements	
High	**F1**	Update file replicas
	F2	Update DB replicas
	F3	Use a lazy protocol
	F4	Use a quorum value for propagating updates
	F5	Provide update status of replicas
	F6	Provide status of running update operations
	F7	Work independently of a RMS
	F8	Provide subscription mechanism for datasets
	F9	One UP protocol for all update operations
Low	**F10**	Users/Admins set protocol for each dataset type
	F11	Users/Admins set protocol for each VO
	F12	Users/Admins set protocol for each dataset
	F13	Users/Admins set protocol for each operation
	F14	Publish information for Monitoring Systems
	F15	Users/Admins set update mechanism
	F16	RCS integrated with a RMS

Table 6.1: Functional requirements.

- *Notes:* The Grid environment is highly dynamic: we have to consider that, at certain times, some replicas might not be available. This non-functional requirement introduces some issues in the main functional requirements, that is Requirement **F1** and **F2**. We analyse this topic later, when we study and specify the use cases (Section 6.2.3.2).

NF3 The RCS shall be designed as a general service, suitable for different Grid architectures.

- *Priority:* High.
- *Notes:* We will use the WLCG/EGEE Grid infrastructure as a specific case but the design should be kept as general as possible to make possible the integration in different Grid architectures.

NF4 The RCS shall provide different command line interfaces suitable for different kinds of clients (i.e., separating user commands from administrative and configuration ones).

- *Priority:* High.

NF5 The RCS shall use secure communications.

- *Priority:* High.
- *Notes:* Security standards must be used; the GSI is the common security infrastructure used in Grid environments and this is the one we will use for our service.

NF6 The RCS shall use configuration files to set up the services.

- *Priority:* High.
- *Notes:* It is already clear that the RCS will have a distributed architecture with many servers. Configuration files will be used to set up these servers.

NF7 The RCS shall allow users to check the consistency of a replica with a checksum mechanism, to prevent the use of corrupted files. A corrupted file may result from incorrect access (for instance when it has been modified without the RCS control) or simply because of some damage, e.g., after a remote transfer or due to disk corruptions.

- *Priority:* Low.

The non functional requirements are summarised in Table 6.2.

Priority	Requirement	
High	**NF1**	Strictly consistent internal catalogue
	NF2	Cope with disconnected sites
	NF3	Allow for integration in different Grid architecture
	NF4	Command line user interfaces of different kinds
	NF5	Use Grid Security Infrastructure
	NF6	Use configuration files
Low	**NF7**	Checksum mechanism

Table 6.2: Non functional requirements.

6.2.3 Use Case Model

In this section we present a use case model for the RCS. We specify *primary actors*, those who will directly use the RCS services, and *secondary actors*, those who will collaborate with the RCS to realise a specific use case. In Figure 6.2 the diagram with the use cases we found is shown.

Figure 6.2: Use case diagram

6.2.3.1 Actors

The main actor of this model is the **Service Client**: it can be a user that directly uses a command line client interface on its UI machine, a user's job or even another software program or middleware service. For this reason we use the class icon instead of the stick figure.

The actor **Privileged User** is a specialisation of the previous actor, since it is a user with specific rights that allow him to do critical operations like setting and changing the RCS properties. Being a specialisation of the "Service Client" actor, it also inherits the use cases of its parent.

Then there is a possible Replica Management Service that might work in collaboration with the RCS: its task as regards the RCS is to integrate information about dataset consistency provided by the RCS with the one that it directly manages.

The class representing the **Monitoring System**, in charge of collecting information regarding the behaviour of the RCS, is another primary actor.

Time can be seen as another actor in that some operations can be executed periodically; in particular we should note that the *Update file* and *Update DB* use cases are associated both to the Service client and the Time actors. This means that these use casee can be executed by the client, with a specific request, or automatically, with a given periodicity.

As regards secondary actors, we have a **Replica Interface** which provides interface services between the RCS and the replicas.

6.2.3.2 Use cases description

In our case a formal use case specification is not very useful because the system has only two basic use cases: *Update file* and *Update DB*; all the others are ancillary to these two or represent simple methods to retrieve or set properties. Moreover, these use cases depend on complex algorithms and can be further specified in an automatic (triggered by a time event) and a manual one (triggered by a user request). Therefore, we focus our attention on these two use cases.

The use cases *Update file* and *Update DB* can be seen as specialisation of a general use case, *Update dataset*. Both of them can also have a manual and an automatic version. This generalisation is shown in Figure 6.3 where we highlighted the two specifications that we will focus on.

The two sub-cases have many similarities but are also different, not only for the data they work on, but also for the way the clients use them. For both cases we will show a basic scenario with the description of the main steps involved.

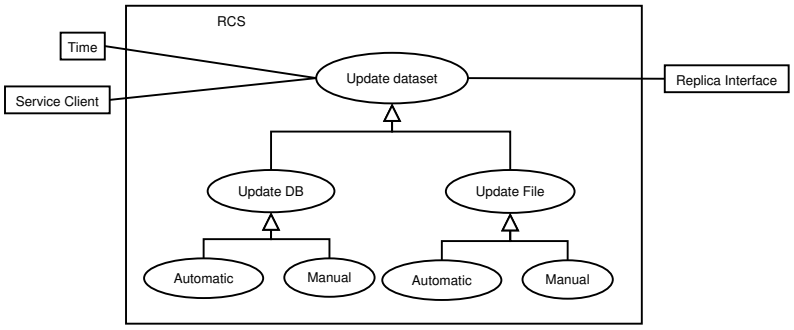

Figure 6.3: Specialisation for the Update dataset use case based on the dataset type

Another specialisation we can do over the *Update dataset* use case is related to the algorithm used. Looking at the classification done in Chapter 3 about optimistic replication, we can derive different use cases from the *Update dataset* one. The Figure 6.4 shows the consistency protocols hierarchy. The choice of the algorithm used is based on application specific requirements.

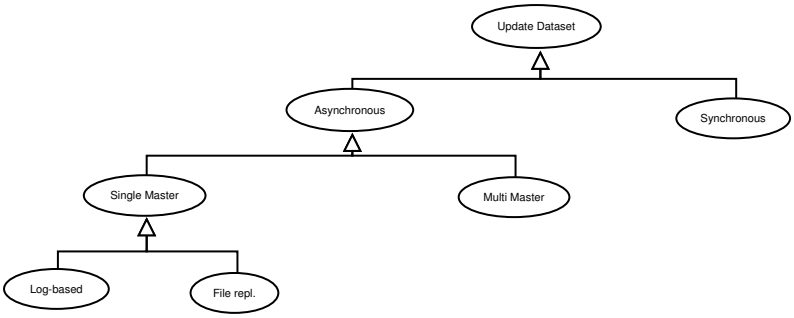

Figure 6.4: Specialisation for the *Update dataset* use case based on the protocol used

In the next section we review two main use cases that will be the focus of our first prototype implementation.

6.2.3.2.1 Automatic DB update with single-master asynchronous log-based push mode protocol Database update is one of the main use cases for the RCS since it satisfies requirements regarding the replication of conditions databases (see Sections 4.3) and Replica Catalogues (see Section 5.2). We now specify all the steps that must be done in order to execute this use case, stating actors involved, pre-conditions, flow of events and post-conditions.

- *Actors:* Time, Replica Interface.
- *Preconditions:*
  1. A logical DB is registered with the RCS together with its replicas.
  2. One of the replicas is the master one, and it is the only one that can be updated by users. The other secondary replicas must be updated by the RCS.
  3. At a certain point in time, the master database is updated and a log is created. A log file on the master database contains the statements that modified the database.
- *Flow of events:*
  1. The use case scenario starts at a certain time and repeats with a given periodicity. The RCS extracts update statements from the log of the master database and creates an update unit.
  2. Database replicas can be hosted on DBMSs of different vendors, with different SQL dialects. The RCS before propagating the update unit must elaborate it to make it suitable for the replica it will be applied to.
  3. The RCS propagates the update unit to the secondary replicas. The update unit is stored until every replica has correctly received and applied it. Replicas are distributed geographically and by means of unreliable links: this implies that the RCS must deal with cases where some replicas are not available during an update operation. An update operation must be able to complete and deal with the disconnected replicas in a flexible way, with the possibility to use a quorum value.
  4. At each replica site the update unit is applied to the database replica using the Replica Interface.
- *Postconditions:*

1. All the replicas of that DB, after a variable amount of time[1], are synchronised.

6.2.3.2.2 Update file with single-master asynchronous file replacement push mode protocol This is the main use case for flat files.

- *Actors:* Service Client, Replica Interface.
- *Preconditions:*
  1. A logical file is registered with the RCS together with its replicas.
  2. A Service Client modifies a replica creating a new version of the file.
- *Flow of events:*
  1. To update all the secondary replicas the Service Client requests an update file operation to the RCS, providing the location of the new version of the file.
  2. The RCS finds all the replicas and propagates the new version of the file. The same consideration about disconnected sites for the previous scenario applies here as well. An update operation must be able to successfully execute despite disconnected nodes.
  3. At the destination sites, the new file replaces the old one.
- *Postconditions:*
  1. All the replicas of that file, after a variable amount of time are up-to-date and synchronised.

6.3 Analysis

In this section we present some analysis concept used to design the main packages and classes of the system. This analysis work is important to clearly state the system domain and to refine and help the maintenance of the requirements. In Section 6.1 part of this work has already been done, and this has helped us to define the requirements unambiguously. However, in this phase we try to formalise more the concepts exploiting the use of class diagrams. In preparing the analysis classes we focus on

[1] It depends upon the laziness of the used protocol and the availability of the replicas.

functional requirements; non-functional requirements will be dealt with during design and implementation.

6.3.1 Analysis classes

As described in Section 6.1 the term dataset can be used as a generic term that refers to the data we deal with: it encompasses structured and unstructured entities. We consider databases as a kind of structured dataset: database replicas are updated using high-level commands, mainly sequences of SQL commands. On the other side, as unstructured dataset, we consider flat files that are modified using byte-oriented I/O operations and that are updated with file replacement or binary difference mechanisms. This classification is shown in Figure 6.5.

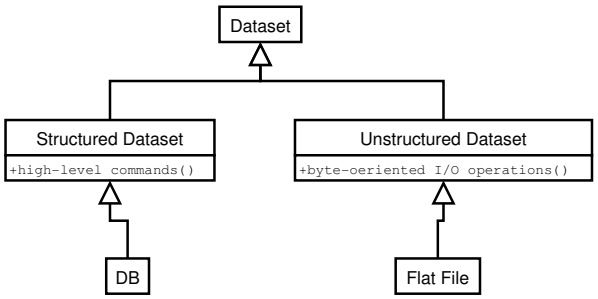

Figure 6.5: Analysis classes for Dataset generalisation

During the domain analysis we already discussed some concepts like replication, logical and physical files. Now we try to further clarify these concepts building some analysis classes that will help us in approaching the design and the implementation phases.

As regards file replication, a Logical File Name (LFN) is an identifier associated to GUID and Replicas. Replicas are identified with a Physical File Name (PFN), have their own contents and are stored in Storage Elements. Storage Elements are of different types and provide different access methods. The PFN[2] holds the information needed to access a given replica: typically this information will include the address of the SE holding the replica, a pathname within the SE's file system (or some logically equivalent information), and possibly an access protocol supported by the SE.

The property of consistency for a set of replicas can be defined as follows:

[2]This term occurs in the literature with slightly different meanings. We use it as a synonym of SURL, that we saw in Section 5.2.

For any pair of replicas of the logical file, their contents are equal.

The above definition can be expressed as a constraint on the association between GUID's and replicas:

{(a.isTypeOf(Replica) and b.isTypeOf(Replica) and a.id = b.id) implies a.contents = b.contents}

Figure 6.6 summarises all these concepts.

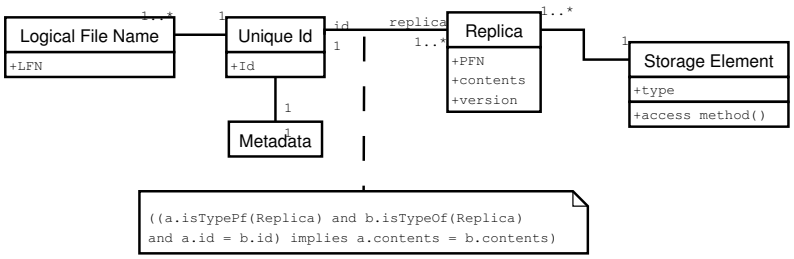

Figure 6.6: File replication concepts.

For databases an analogous analysis can be done. A logical identifier, Logical Database name (LDB) can be used to identify a set of possibly heterogeneous (different vendors) replicated databases. The schemas of the replicas must be equivalent, and when the replicas are consistent the same query should return the same results in each replica. Database replicas hold a master/slave attribute that has a meaning according to the protocol used to maintain the consistency. They are also characterised by a version number. When they are consistent their version number should be the same. In Figure 6.7 these concepts are represented.

Figure 6.8 shows a static view at a certain point in time in a scenario where the LFC (see Section 5.2.1.4) database is replicated with an Oracle master database and two MySQL slave replicas. The snapshot highlights that at a certain instant the replica named "slave2" is stale compared to the other slave replica "slave1". This staleness is possible since the algorithm is lazy: the stale replica could be temporarily unreachable or might have simply not yet received the update.

At this point, the similarities between the two cases can be exploited, and the two diagrams for files and databases can be unified using the dataset concept and generalisations. Figure 6.9 shows the unified scenario in terms of datasets.

The term *update operation* is present in both the use case description for file and database update. We now look for analysis classes considering the *update dataset* use

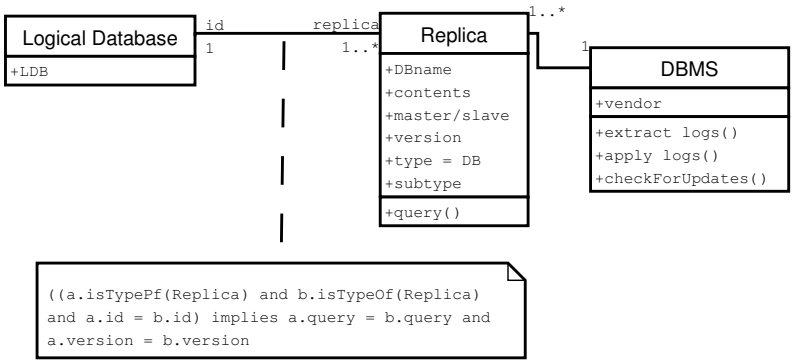

Figure 6.7: Database replication concepts.

case, which includes both files and database synchronisation. An update operation is requested by a Service Client or triggered by a Timer, and performed by the RCS. It involves the synchronisation of all the replicas of a given logical dataset which are distributed over a WAN and connected with unreliable links. To implement the update propagation protocol, an update operation must act on dataset replicas using the Replica Interface. An update operation is characterised by an unique identifier, and it operates according to a specific update propagation protocol to deliver updates and to a specific update mechanism to apply these updates. It also holds status information that can be retrieved by Service Clients that issued the update request. A quorum attribute is used in order to decide whether to execute or not an update propagation. In Figure 6.10 a class diagram that summarises these concepts is shown; we can notice that the *Update Operation* class plays an important role being associated with all the other analysis classes.

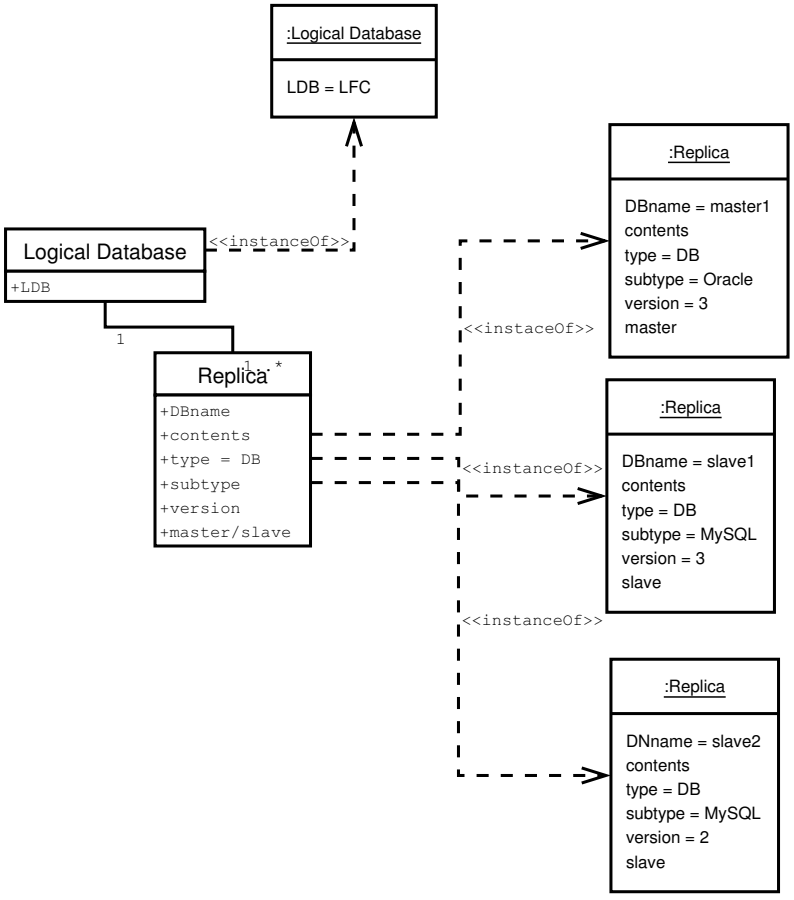

Figure 6.8: Class diagram for DB replication

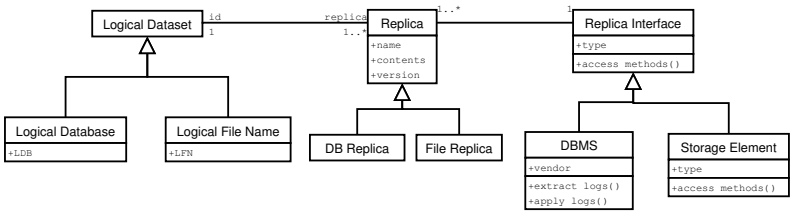

Figure 6.9: Dataset replication concepts.

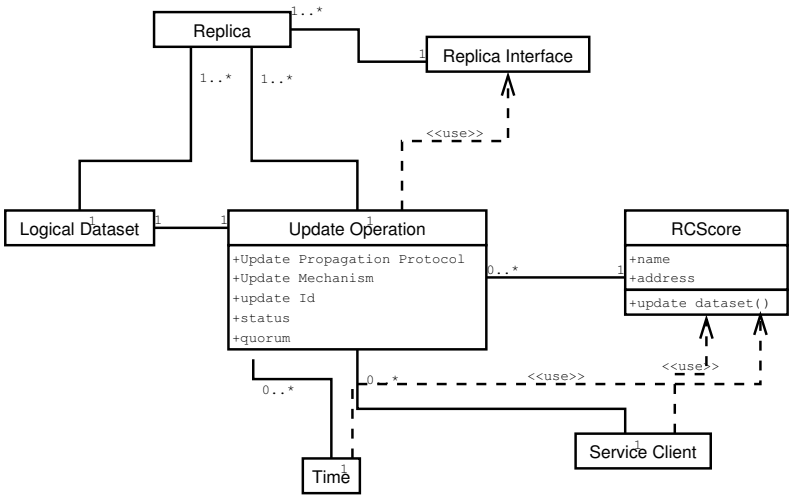

Figure 6.10: Analysis classes for the Update Dataset use case

6.3.2 Use case realisation

Use cases realisations are important during an analysis work in that they explain, through the use of analysis classes, how a specific use case is realised by the system.

In case of databases, we decided to start with a simple single-master scenario. where the update propagation is triggered by the Time actor. For flat files the scenario is different in that a client can retrieve and modify a replica and then submit a synchronisation request.

6.3.2.1 Automatic database update with single master asynchronous log-based push mode protocol

In case of database synchronisation, the class diagram of Figure 6.10 can be enhanced with two new classes. The result is shown in Figure 6.11

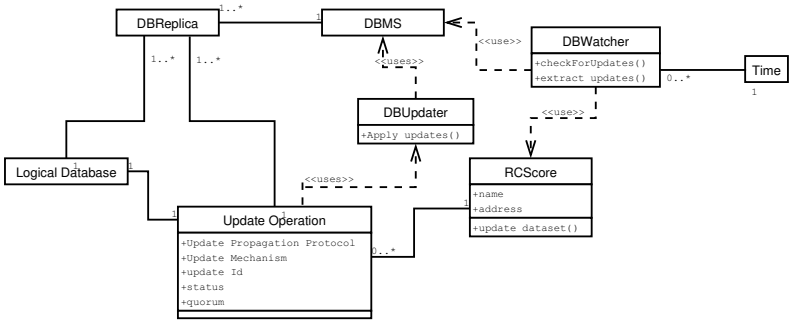

Figure 6.11: Analysis classes for the synchronisation of databases

The class `DBWatcher` is the entity whose task is to monitor the master database looking for updates. In case of an automatic procedure, the `DBWatcher`'s activity is triggered by a timer so that it checks the master database with a specific periodicity. The `DBWatcher` uses the DBMS in order to find and in case extract updates from the master database. The class `RCScore` is contacted by the `DBWatcher` in order to start an update propagation phase. RCScore uses the `Update Operation` class to manage each single update propagation task. A class `DBUpdater` is in charge of applying the updates to slave replicas, through the DBMSs that manage the database replicas.

The realisation of this use case is shown in Figure 6.12 through a sequence diagram where the main analysis classes are involved. The Timer triggers a check operation executed by the `DBWatcher` class to find out whether the master database has been updated since the last check. In case updates are found on the master database,

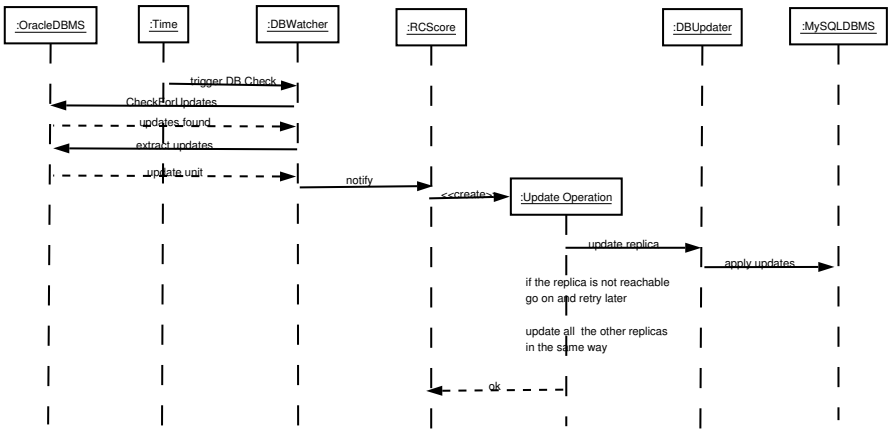

Figure 6.12: Sequence diagram for the synchronisation of databases

the SQL statements that modified the database are extracted from the master database and put in an update unit. The RCScore is then contacted in order to manage an update propagation task. This is done through the creation of an an Update Operation object responsible for the implementation of the update propagation algorithm. The `Update Operation` object contacts every secondary replica (slave database), and, using the `DBUpdater` class, applies the updates.

6.3.2.2 File synchronisation with asynchronous single master push based protocol

As regards the file synchronisation, the class diagram in Figure 6.10 can be refined into the specific model for flat files. The result is shown in Figure 6.13.

In this scenario the actor requesting the update operations is the Service Client. The Update Operation class, responsible for the implementation of the update propagation protocol, uses the services offered by the Storage Element in order to update replicated files. In Figure 6.14 a sequence diagram describing this scenario is shown. Assuming a logical file is already registered into the system, together with its replicas, stored at several Storage Elements, the Service Client can retrieve a copy of a file, modify it, and submit an update request to the RCScore specifying the location of the modified copy and the logical name of the file. At this point an Update Operation object is created in order to manage the update of each replica of that logical file. All the Storage Elements that hold a replica of that file are contacted to replace

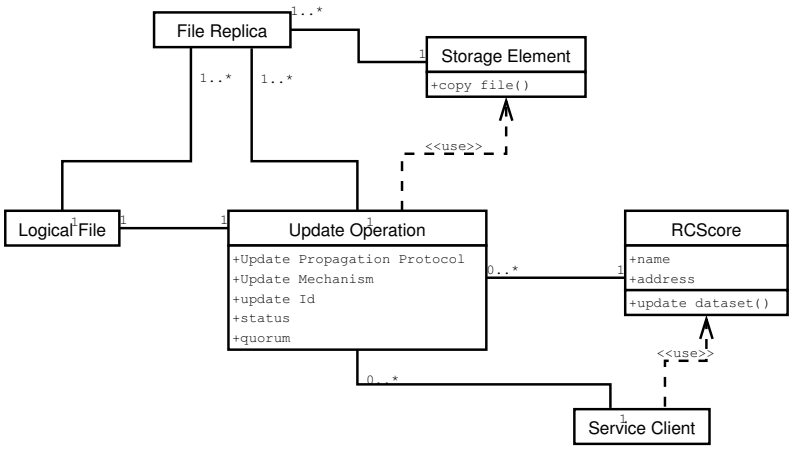

Figure 6.13: Class diagram for the synchronisation of replicated files

the old copy of the file with the new one.

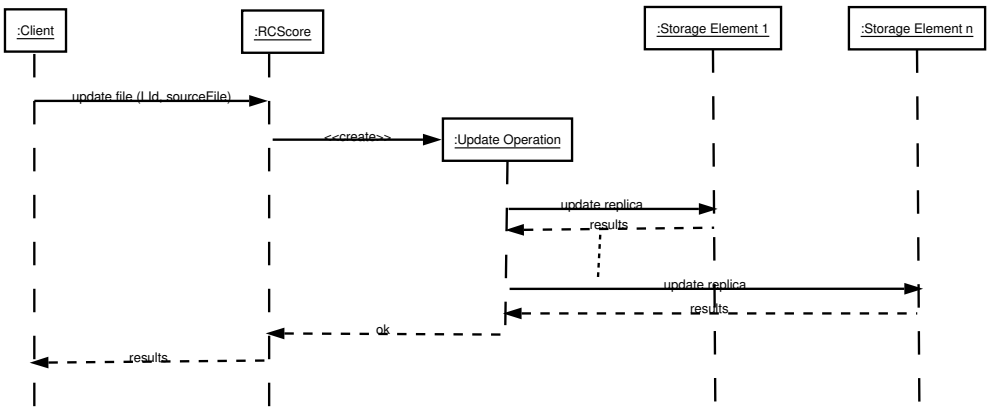

Figure 6.14: Sequence diagram for the synchronisation of flat files

6.4 Design and Implementation

In this section we investigate the architecture of the system, considering also non-functional requirements. Analysis classes will be detailed, and new classes will be added to the analysis model. To make this section more readable, we focus on the main use case for the synchronisation of databases also because this use case is the most urgent one and the one for which detailed tests have been performed (see Chapter 7).

6.4.1 Nodes and network configuration

The RCS must be designed as a distributed system. In this section we focus on the specification of the nodes and network configuration. In Figure 6.15 the nodes and the connections that form the RCS architecture are shown.

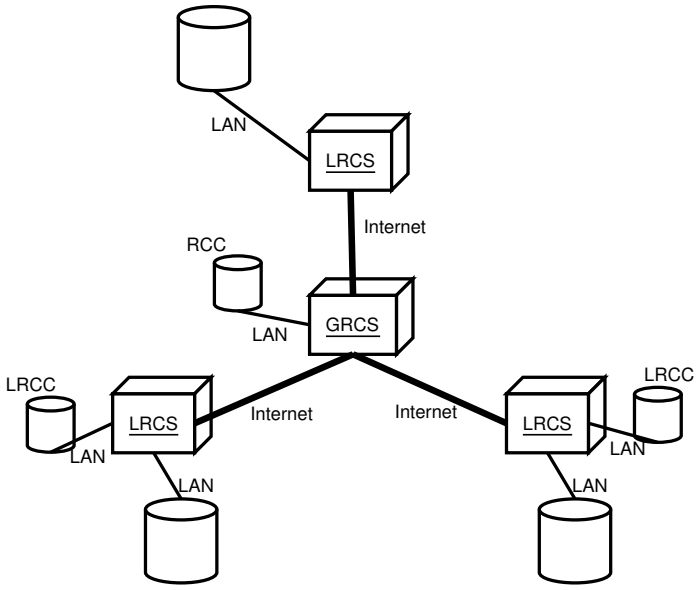

Figure 6.15: Network configuration

The RCS has a distributed architecture with two types of nodes (servers): the Global RCS (GRCS) and several Local RCS (LRCS). Each server uses a relational database to store persistent status information.

The GRCS is the main component of the system and the one that can be contacted directly by clients in order to manage an update operation. The GRCS also controls all the LRCSs that are part of the RCS architecture.

The database used by the GRCS is called *Replica Consistency Catalogue (RCC)* while the database used by the LRCSs are called *Local Replica Consistency Catalogues, (LRCCs)*. Particular attention is required on these catalogues, whose information is critical for the activity of the overall service. A fail-over solution must be implemented on these databases.

LRCS nodes are responsible for the management of the database replicas on behalf of the Replica Consistency Service. An LRSC can be deployed at a master site (the site holding the master database) or at a slave site (a site holding slave database replicas). Regarding the site where they are deployed, we usually refer to master or slave LRCSs. Among their tasks, the most important are the monitoring of the master database and the extraction of update statements in case of a master LRCS, and the application of updates in case of a slave LRCS.

6.4.2 Subsystems decomposition

Identifying subsystems is useful to decompose the model into several pieces on which we can often work in parallel. Subsystems are conceptually separated parts of the system that perform specific tasks. In our case the subsystems decomposition is tightly bound to the decomposition into network nodes.

In Figure 6.16 the main subsystems that form the RCS software are shown.

The subsystems GRCS and LRCS have a similar structure, with many internal subsystems in common. The `Core` subsystem is where the logic of the application is implemented, mainly tasks concerning the update propagation protocol and all the other tasks that derive from the functional requirements (see Section 6.2.1). The `Server` subsystem includes the server code and scripts used to start, stop and restart the server. The `Communication` subsystem is the one where the code related to client-server and server-server communications is placed. The `Security` subsystem is concerned with authentication, privacy, integrity and authorisation in these communications. The `Client` subsystem holds the client programs used to communicate with the servers. Finally, the configuration subsystem is responsible for the management of configuration files and scripts that will be used to set up and start the servers.

Within the LRCS subsystem we can identify two additional components that can be developed as independent subsystems: the `DBWatcher`, to monitor a mas-

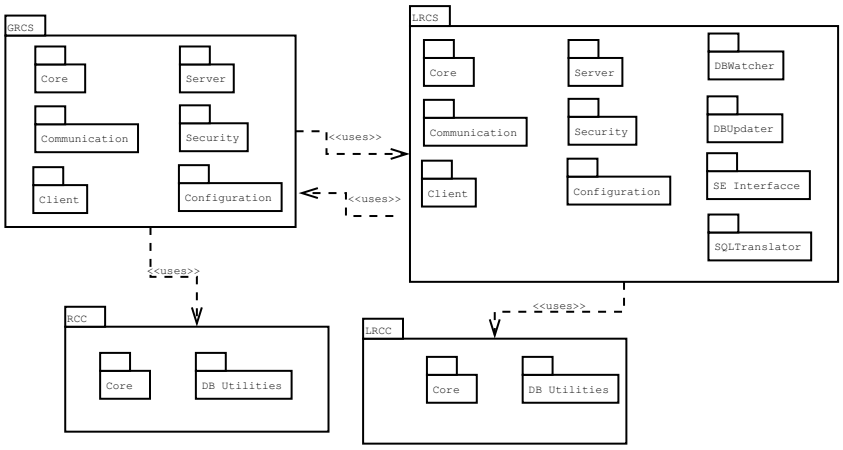

Figure 6.16: Main subsystems of the RCS

ter database in case of database synchronisation, and the `DBUpdater`, in charge of the application of the updates to secondary replicas. In case of file replication instead, an SE Interface subsystem will interface the LRCS with the SE using specific protocols. We used also two different subsystems to group the software concerned with database access, one for the RCC (used by the GRCS) and one for the LRCCs (used by the LRCSs). These subsystems are made of a Core part, to implement all the queries to the database and a DB Utilities part, where we will put utility code for the management of the databases.

6.4.3 Subsystems in details

In this section, we refine the analysis classes previously found (see Section 6.3.1) and enhance that model with some details about analysis classes and the insertion of new classes. The subsystem decomposition will help us to define what are usually called design classes, refined version of the analysis classes, and to logically place them into the appropriate subsystems. We decided to provide details about the subsystem as much as it is required to understand the logic of the service, especially for what concern the use case for synchronisation of databases. For more details the repository of the CONStanza project can be browsed [2].

6.4.3.1 GRCScore

In Figure 6.17 the design classes for the subsystem `GRCScore` are shown.

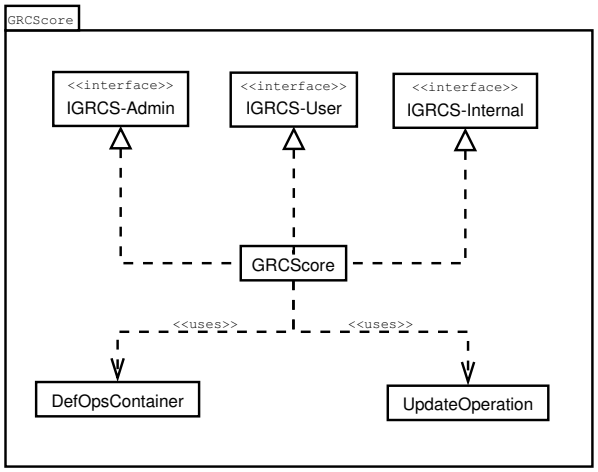

Figure 6.17: Design classes for the `GRCScore` subsystem

Three different interfaces are implemented by the `GRCScore` class. An administration interface, `IGRCS-Admin`, exposes administration operations that require certain privileges, and will be normally executed by the service administrator (Privileged User in the Use Case model of Figure 6.2). A user interface, `IGRCS-User`, exposes methods that can be called by normal service users, like starting an update propagation. Finally, an internal interface, `IGRCS-Internal`, is provided for those methods that will be requested by LRCSs. The `GRCScore` class uses two important classes. An `UpdateOperation` class is responsible for the management of an update propagation

phase. The class `DefOpsContainer` is a class used to manage update operations that have not successfully completed due to some LRCS faults. This container will store uncompleted operations for them to be re-executed.

Let us start to look into the details of these classes.

6.4.3.1.1 IGRCS-Admin This interface exposes methods used by privileged users to set up important RCS parameter and perform critical operations. The main methods of this interface are:

- `int setUPProtocol(ProtEnum prot);`

  It is used to set the propagation protocol.

- `int setUpdMechanism(UpdMechEnum mech);`

  This method allows privileged user to set up the update mechanism.

- `int setQuorum(int quorum);`

  It is used to set up a quorum value.

- `int subscribeLRCS(URI addr, string name);`

  This method is used to subscribe an LRCS to the GRCS. The GRCS uses a subscription method to keep track of the available LRCS that are part of the RCS architecture. The GRCS identifies an LRCS using the server URI plus a name.

- `int unsubscribeLRCS(URI addr, string& name);`

  This method can be used in order to unsubscribe a specific LRCS.

6.4.3.1.2 IGRCS-User This interface provides basic services that non-privileged users can request. In what follows we cite the main ones:

- ```
  int updateDB(LId ldb, URI logFile, int quorum,
                  Version version, UpdId& res);
  ```

  This is the method used to request an update of database replicas. It is used in the "manual" scenario, where the update is triggered by a client request. The request must specify a logical database, the location of a log file containing the SQL statements to be applied to secondary replicas, a quorum value to be used for this update propagation and a version number to be used to avoid
  ```

conflicts. An update Id is returned to the user that can use it to check the status of the operation. To satisfy such a request the `GRCScore` creates an `UpdateOperation` object and executes it.

- `int updateFile(LId lfn, FileURL src, int quorum, Version ver, UpdId& res);`

  This is the method used to update file replicas. It is the same as in the case of database update but this time instead of a log file we use as "update source" a file, that will replace the stale replicas.

- `int getStatus(UpdId id, UpdStatus& res);`

  This is the method used to check the status of an update operation. The status can be one of: CREATED, RUNNING, FAILED, UNCOMPLETED, DONE, NA. The NA status is returned when the Update Id provided is not valid.

The rest of the methods of this interface can be found in the CONStanza repository.

6.4.3.1.3 IGRCS-Internal This interface groups the method requested by LRCSs.

- `int ackUpdate(UpdId id, UpdStatus st);`

  It is used to send an acknowledgement about a requested updated operation.

- `int refreshLRCS(URI addr);`

  It is used to refresh information stored by the GRCS about a specific LRCS.

6.4.3.1.4 DefOpsContainer This is the class used to manage operations that are not completed due to some unavailable LRCS. These operations will be re-executed at a later time. This class holds a container plus methods to interact with it. The container is a map that associates to each LRCS a queue of operations that must be completed on the same LRCS. In Figure 6.18 an example of the structure of the container is shown. During an update propagation phase, when an LRCS is unavailable to receive an update, the GRCS (more precisely an `UpdateOperation` object) will insert a reference to the operation into the queue of that LRCS. Later, when the LRCS will be available, this operation can be re-executed. The main methods to access the container are:

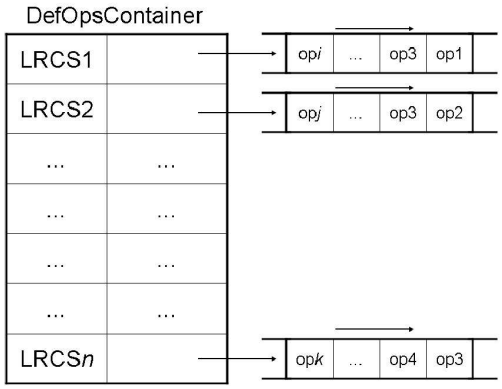

Figure 6.18: An example of the `DefOpsContainer` structure

- `int pushOperation(LRCSaddress addr, UpdateOperation* pOp);`

  It is used to insert an operation into the queue of a given LRCS.

- `int nextOperation(LRCSaddress addr, UpdateOperation*& pOp);`

  It is used to select the next operation to be processed from the queue of a given LRCS. It does not remove the operation from the queue.

- `int popOperation(LRCSaddress addr, UpdateOperation*& pOp);`

  It is the same as `nextOperation` but this time removing the operation from the queue.

- `int remOperationFromQueue(LRCSaddress addr, UpdId id);`

  It is used to remove a specific operation from the queue of an LRCS.

- `int emptyQueue(LRCSaddress addr);`

  It is used to empty the queue of an LRCS.

- `int print();`

  It is used to print the status of the container: for each LRCS all the operations in its queue will be listed.

- `int isEmptyQueue(LRCSaddress addr);`

  It is used to verify that the queue of an LRCS is empty.

6.4.3.1.5 GRCScore The `GRCScore` is the most important class of the RCS architecture since it contains the logic of the GRCS, the main node. It implements the interfaces `IGRCS-Admin`, `IGRCS-User` and `IGRCS-Internal`. It has a reference to the RCC and to the `DefOpsContainer`, that are singleton objects, used respectively to store status information and uncompleted update operations. It also has a container to store the addresses of the LRCSs that have been correctly subscribed, plus objects to hold the type of UP protocol and Update Mechanism used. Figure 6.19 provide a view of the main data member of the `GRCScore` class. It does not show the imple-

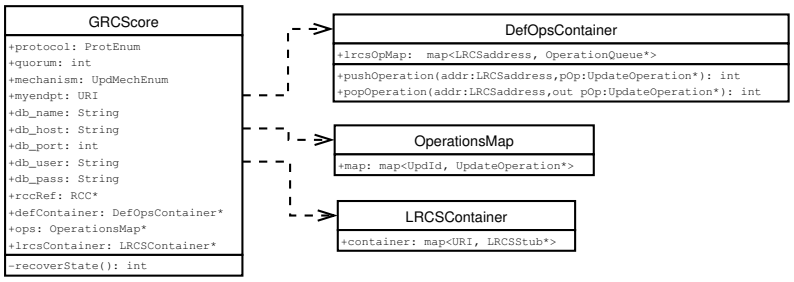

Figure 6.19: `GRCScore` class

mentations of the interfaces to make the figure more readable. As we can see, the main properties that define how the synchronisation of replicas is performed are:

- protocol: it defines the protocol used for update propagation, the UP protocol; it can be one of the protocol names defined in the `ProtEnum` enumeration. In our first prototype implementation only one protocol is defined, labelled `ASYNCH_ONE_MASTER`.

- mechanism: it defines the mechanism used by LRCS for replica update. As we saw in Chapter 3, several mechanisms can be used, like total file replacement, log-based, or binary difference. In our first prototype implementation log-based is used for database synchronisation while total file replacement is used for flat file synchronisation.

- quorum: this is an integer value which defines the minimum number of LRCS that must be available in order to execute an update propagation. This value

can be set as a global RCS property or can be given to an update command line client as argument to be used only for that single operation.

Then we have data to define the access to the database used to store the RCC and a field to keep a reference to it. The `defContainer` field holds a reference to a `DefOpsContainer` that is used to save deferred update operations as explained in 6.4.3.1.4. The `ops` field is used to save information about running update operations and the `lrcsContainer` points to a container that keeps information about all the LRCS that have been correctly subscribed to the GRCS, holding a pointer to the Stubs used for remote calls to each of them.

6.4.3.1.6 UpdateOperation The `UpdateOperation` class is used to execute the UP protocol: every update request triggers the creation of a new `UpdateOperation` object that takes care of implementing the update propagation phase. The `UpdateOperation` class has already been outlined as an analysis class (see Figure 6.10). In Figure 6.20 more details are given.

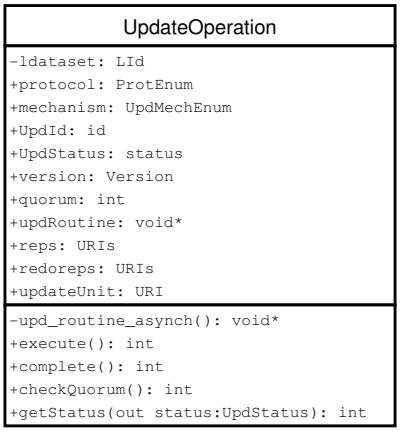

Figure 6.20: Update Operation class

The main methods are:

- `int execute(double& Status);`

  It is the method used to execute the operation, that is to execute the propagation protocol.

- `int getStatus(UpdStatus& status);`

It is used to retrieve the status of the operation, which changes during its execution.

- `int complete(dUpdStatus& Status);`

  It is used to re-execute the operation when the first execution has not been completed due to some unavailable LRCS. In this case a reference to this operation is kept inside the DedOpsContainer.

- `int checkQuorum();`

  It used before executing the operation, in order to check whether enough LRCSs are available for receiving the updates.

In Figure 6.21 the update propagation phase executed by an `UpdateOperation` object is shown.

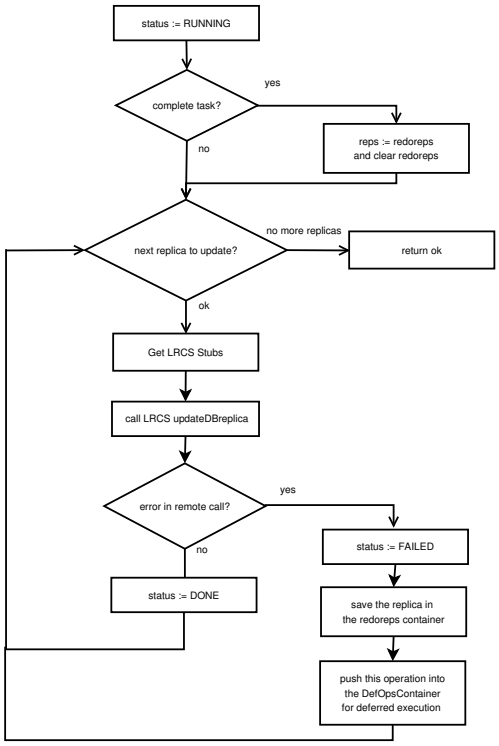

Figure 6.21: Update Propagation phase

An UpdateOperation execution is triggered by the GRCS only when the quorum check has been successfully passed. When calling the execute method on the UpdateOperation object, the object has already been given the list of replicas that need to be updated and the location of the update unit. During the processing of an update propagation the status of the operation is kept both in memory and into the RCC database.

The same procedure is used by an UpdateOperation object to perform a "first execution" or a "complete-task". A first execution is the normal activity, while a complete task is the execution of an update propagation done after a failure, when the operation is saved into the DefOpsContainer. Thus, at the beginning, it is important to state whether the current execution is a complete task or not. In case of a complete task the set of replicas used for the current execution is retrieved from the redoreps container, where the replicas not updated in the previous executions are saved. Then, for each replica to be updated, the UpdateOperation gets the Stub of each LRCS to contact and issues a remote call to them to execute the updateDBreplica method. If the remote call is successfully executed, then the flow of events goes on to pick up the next replica to be updated, otherwise the operation inserts details about the replica in the redoreps container, and some information into the DefOpsContainer in order to be re-executed on the failed replicas later on.

6.4.3.2 Communication subsystems

For the Communication subsystems, we decided to use the tool gSOAP [86] for the creation of web services. Using this tool we can easily create web services without developing the SOAP/XML infrastructure. Stub and skeleton objects are automatically created by the gSOAP compiler starting from a description of the service and its remote call included in a C++ header file. Together with them a set of utilities are also created for the de/serialisation of C++ objects into XML. We will consider these utilities as part of the stub and skeleton classes. Two header files for the GRCS and the LRCS web services will be provided. Using the gSOAP stub and skeleton compiler, skeleton classes (WGRCS and WLRCS) and stub classes (GRCSproxy and LRCSproxy) will be generated. The Communication subsystem of the GRCS and LRCS will use these gSOAP classes to implement the remote method invocation as depicted in Figure 6.22.

We don't go through the details of this package since they are mainly implementation details that can be skipped in order to understand the logic of the service.

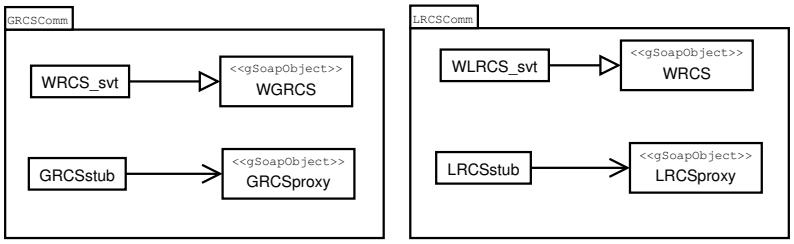

Figure 6.22: Design classes for the `LRCSComm` and `GRCSComm` subsystems

6.4.3.3 Security subsystems

For what concerns the security of our system, the solution provided by the Globus Toolkit (see Section 5.2.2), GSI, is the standard in Grid environments. Moreover, using gSOAP for the implementation of the communication mechanisms, we can easily integrate the GSI into our architecture using a gSOAP plug-in called CGSI [1]. Using CGSI the communication between GRCS and LRCSs, and between Clients and the GRCS are secure, meaning that they support authorisation, authentication, privacy and integrity. A more detailed description on how the GSI has been integrated into the RCS has been presented in [74]. Details on how to set up the certificate and the private keys for the RCS servers are included into the RCS User Guide [2].

6.4.3.4 Configuration subsystems

The `Configuration` subsystem is present in both the GRCS and the LRCS subsystems. It contains configuration files needed to setup the servers. It also contains two classes used to parse the configuration files, in particular a lexical analyser and a parser. In order to build the lexical analyser and the parser we use Flex and Bison [23]. A `TokenFile` file will contain all the lexical patterns and will be used by the Flex compiler to build the lexical analyser (`GRCSconfLexer`). Using a syntax file (`SyntaxFile`) then, the Bison compiler will generate the parser (`GRCSconfParser`). Finally, using the lexical analyser, the parser will be able to read the configuration file and setup properties that will be used by the `GRCScore` package. In Figure 6.23 the `Configuration` subsystem for the GRCS package is shown. An analogous diagram can be built for the LRCS.

Configuration files are text files with key-value pairs. These pairs are then logically grouped into sections. In the GRCS configuration file, three sections can be used: GRCS, LOGGING and RCC. The GRCS section contains information like the

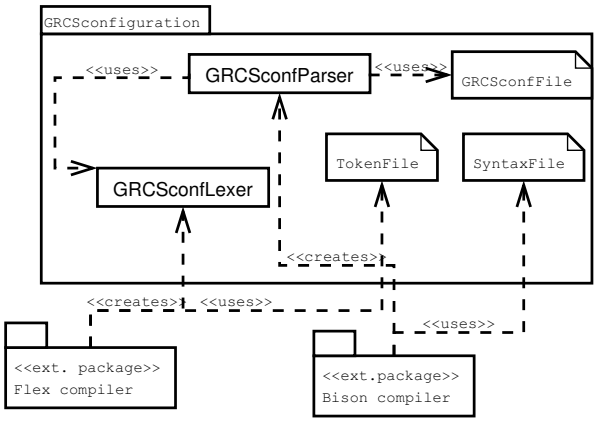

Figure 6.23: Design classes for the `GRCSconfiguration` subsystem

server endpoint, the protocol and the quorum used and a name to identify the GRCS. The LOGGING section is used to group information related to the logging activity like the filename and the directory to be used to store log files, and the level of severity of the logging activity. For what concerns the technology to be used for logging the activity of the RCS, we decided to use a C++ library already available, Log4cpp [45].

Configuration files for the LRCSs have a similar structure. The role that the LRCS server will play (master or slave) is decided using the appropriate LRCS configuration files. Two types of configuration files exist, one for a master LRCS (`lrcs.master.conf`) and one for a slave LRCS (`lrcs.slave.conf`). Which configuration file will be used is specified in another file that is automatically sourced by the server scripts. This file is called `lrcs_profile`. In this file we can also specify whether we want to use the GSI security or not, which level of debug will be used, and the hostname and port to be used. An example of `lrcs_profile` file is the following:

```
#!/bin/bash
#
# description: LRCS server profile script

LRCS_HOST="pcgridtest3.pi.infn.it"
LRCS_PORT=8091
LRCS_DEBUG="1"
LRCS_GSI="ON"
```

```
LRCS_CONF="lrcs.master.conf"
```

In this case the LRCS server will act as master, using the information recorded in the file `lrcs.master.conf`. In case of database synchronisation, the LRCS configuration files have a section to hold specific information about the database to manage, like the vendor, the account to be used to access it, and the name of the database. In Appendix A some examples of configuration files for the GRCS, LRCS master and LRCS slave are given.

6.4.3.5 LRCScore

The `LRCScore` subsystem is where the logic of the LRCS server is implemented. An LRCS node can act as a master or as a slave, but this distinction is done only at configuration time, before starting the server (See Section 6.4.3.4). In case of master LRCS, the `LRCScore` module will mainly use the `DBWatcher` component in order to monitor a master database; this task is explained in detail in Section 6.4.3.9. In case of slave LRCS, the `LRCScore` module will mainly receive update requests from the `GRCScore` module, and specifically from an `UpdateOperation` object which manages an update propagation operation. The module `DBUpdater` is then used to perform the actual replica update, using the selected replica update mechanism. The update of a database replica is explained in Section 6.4.3.7. In Figure 6.24 the main structure of the `LRCScore` module is represented, with a level of detail enough to understand the following sections.

An LRCS node can manage several master and slave replicas. Although a logical database can have at most one master replica[3], an LRCS can manage master replicas of different logical databases. An LRCS can also manage several slave replicas, and in this case they can belong to the same logical database. In order to do this, the `LRCScore` uses some data structures to keep track of the `DBUpdater` (`UpdatersMap`) and `DBWatcher` (`WatchersMap`) components it uses.

Another important data structure, fundamental for ensuring fault tolerance on the slave database replicas, is the map of pending updates (`PendingsMap`). When a database update cannot be applied to the slave database (for example when the DBMS server is down), it is saved in the `PendingsMap` and it can be re-applied later on when the connection with the database is re-established. The procedure used to apply pending updates ensures that the correct order of the update is respected. In the following sections, after presenting the `DBUpdater` component, we provide more details on how the database replica update phase is implemented.

[3] Multi-master replication is not yet supported.

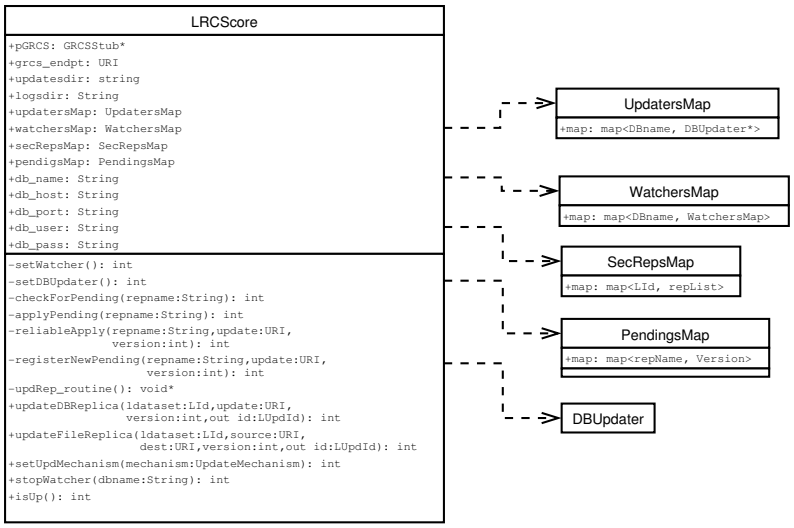

Figure 6.24: Design classes for the LRCScore subsystem

6.4.3.6 DBUpdater

The `DBUpdater` subsystem is composed of a single class, the `DBUpdater` class. It is used by the `LRCScore` subsystem to apply update units to secondary replicas. In case of an Oracle to MySQL replication the `DBUpdater` will use the MySQL client program in order to perform its task. In Figure 6.25, the `DBUpdater` subsystem and its dependencies is shown.

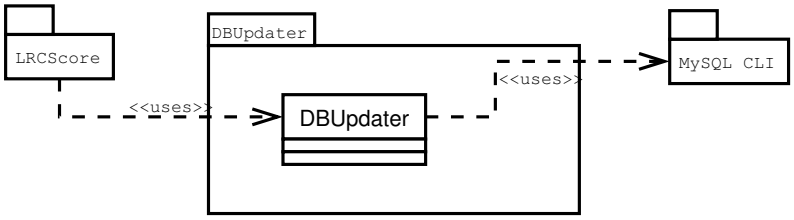

Figure 6.25: Design classes for the `DBUpdater` subsystem

In Figure 6.26 the basic structure of the `DBUpdater` class is shown. It is worth noting that also the `DBUpdater` keeps track of the version of the database it manages. This version corresponds to the number of update units correctly applied. A `checkVersion` method is used to check if an update unit can be correctly applied;

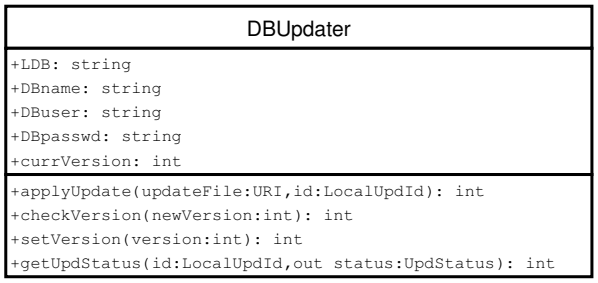

Figure 6.26: Design classes for the DBUpdater subsystem

due to networks interruptions or delays it can happen that the order of two update requests coming from the GRCS is inverted. The version mechanism implemented by the DBUpdater ensures that the correct order of application of the updates is respected.

6.4.3.7 Database Replica Update

The update of a database replica is done in a non-blocking way by the LRCScore method updateDBreplica. A thread is spawned and an Id number is returned. Before starting the thread, the updateDBreplica method retrieves the list of secondary replicas to update (from the map of secondary replicas) and passes it as argument to the thread. The multithreaded implementation has been done to speed up the update propagation phase; an updateDBreplica call is assumed successfully executed as soon as a thread is spawned correctly and an Id is returned.

We now look at the body of the thread that implements the actual update of the database replica. The flow of events is shown in Figure 6.27. The Database Replica Update phase starts with a GridFTP file transfer, where an LRCS slave node retrieves the update unit. The location of the file has been specified by the updateDBreplica call issued by an UpdateOperation during its update propagation phase. The update unit is usually located in a standard location at the master site (usually /tmp/logs), where the DBWatcher component stores it after the translation operation (see Section 6.4.3.9). A GridFTP server must be enabled on the master site to serve the request issued by the LRCS slave, that saves the file in a standard location (usually /tmp/updates[4]).

[4]The directory where logs and update units are stored can be changed before starting the LRCS server using the LRCS master or slave configuration files, explained in Section 6.4.3.4.

Since more database slave replicas can be managed by an LRCS server, the steps shown in Figure 6.27 are repeated for each replica. A reference to the appropriate `DBUpdater` is retrieved from the `UpdatersMap`, and a version check is done in order to respect the correct order of application of the updates. If the version check fails, the update is not applied but saved as pending update into the `PendingsMap`. In this case, an `applyPendings` routine is called which applies the pending updates in the correct order, included the one just saved. If the version check was successful, the update is immediately applied and the version of the replica is incremented. After this, the pending updates map is checked and possible pending updates are applied.

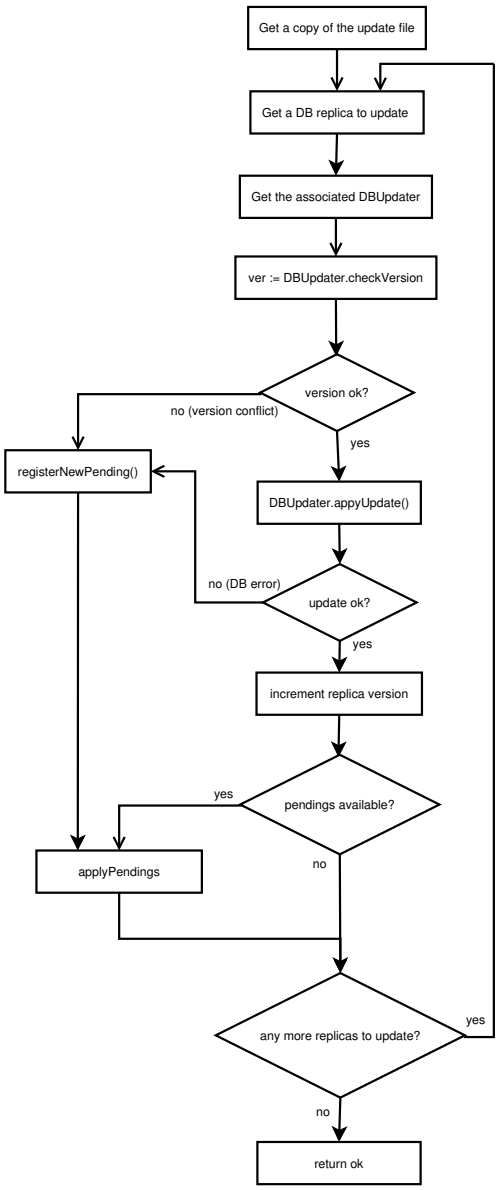

Figure 6.27: Flow of events for the Database Replica Update phase

6.4.3.8 Oracle Log Mining

Before going into the details of the `DBWatcher` architecture, we give some information on how it is possible, in Oracle, to perform the log mining operation. Log mining means analysing the database log information in order to retrieve the operations done on the database. Log mining is usually done to track the usage of the database, or to recover lost data, but it also plays a fundamental role for replicating the database.

In Oracle, the log mining phase is also part of the Streams work flow, described in Section 3.5. Oracle stores log information into two or more *redo log* files. In what follows we assume that we are in a single instance database; multiple instance databases are used in RAC (Oracle Real Application Cluster) environments, and present important differences for what concerns log mining. CONStanza, in the current release, does not support the replication from Oracle to MySQL when the Oracle master database is implemented in a RAC environment.

Redo log files are filled with SQL statements that modified the database (DML plus DDL statements) and other metadata. These data are generally written into the redo log files whenever a transaction is committed, but in certain situation they can be written before the commit. Redo log files are written in a circular fashion: when the current log file fills, the next file is written, and when the last one also fills, the DBMS start overwriting the data present in the first file. The DBMS can work in a special mode, called *ARCHIVELOG* mode, where every time a redo log file is to be overwritten the DBMS makes sure that the file is firstly archived, that is, a copy of the file is saved on the disk. These copies are called *archived redo log files* while the others are called *on-line redo log files*. Different redo log files configurations are possible, each providing a different degree of protection against data loss. We do not go through these details since they are outside our scope.

The *LogMiner* is an Oracle utility that allows you to query the redo log files (both on-line and archived) with an SQL interface. LogMiner operations are available in the `DBMS_LOGMNR` PL/SQL package, and work on the `V$LOGMNR_CONTENTS` view.

To start the LogMiner we have to use procedure `DBMS_LOGMNR.START_LOGMNR` providing some arguments like:

- dictionary option: a data dictionary is used to translate object names stored in an Oracle special format into a human readable form. Oracle recommends to use the option `DICT_FROM_ONLINE_CATALOG` [40].

- redo log file option: when mining the log files the LogMiner needs to know which redo log files to use. Oracle gives you the possibility to provide a list of

redo log files to use, or to ask the LogMiner to automatically and dynamically create such a list by using the option `CONTINUOUS_MINE`.

- time interval: if you want to look for data in a specified time range, you can provide a time interval to the LogMiner as an option, by specifying two values: `starttime` and `endtime`.

Many more options can be given to the LogMiner but these are enough to understand how we used it. Once started the LogMiner with the appropriate options, one can issue queries on the `V$LOGMNR_CONTENTS` view and retrieve data.

In CONStanza we developed an additional PL/SQL package with some utility procedures that can be used to start the LogMiner, but also to test its behaviour. The complete code of the package is reported in Appendix C. In the following we show the procedure that is used by the `DBWatcher` to start the LogMiner:

```
procedure startLogMinerCMine(t1 varchar2, t2 varchar2) is
begin
    dbms_output.PUT_LINE('Starting LogMiner: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

    sys.dbms_logmnr.start_logmnr(options =>
            sys.dbms_logmnr.dict_from_online_catalog
            + sys.dbms_logmnr.committed_data_only
            + sys.dbms_logmnr.no_rowid_in_stmt
            + sys.dbms_logmnr.continuous_mine
            , starttime => to_date(t1,'DD-MON-YYYY HH24:MI:SS')
            , endtime => to_date(t2,'DD-MON-YYYY HH24:MI:SS'));
     dbms_output.PUT_LINE('LogMiner started: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));
end startLogMinerCMine;
```

The procedure takes two arguments, that are used as `starttime` and `endtime` for starting the LogMiner. It uses the `continuous_mine` option, and specifies that only committed transactions should be analysed, since these are the only ones involved in a replication procedure. The `no_rowid_in_stmt` is used to avoid having some metadata in the queries results.

6.4.3.9 DBWatcher

The DBWatcher is a subsystem whose task is to provide the capabilities of monitoring a master database. Monitoring a database means periodically checking its log files and find out when the database is updated. When the DBWatcher finds some updates, it must be able to extract and put them in a file that we will refer to as *update unit* or update file. Putting the updates in a file simplifies the update propagation phase, which can be done with file transfer services already present in all the Grid middleware, like GridFTP(see Section 5.2.2). A GridFTP server can be easily installed on LRCS nodes. In case of an Oracle master database, the DBWatcher component can exploit the services offered by the Oracle LogMiner package (see Section 6.4.3.8). In Figure 6.28 the DBWatcher subsystem with its dependencies is shown. The Oracle LogMiner subsystem is used to monitor the Oracle master

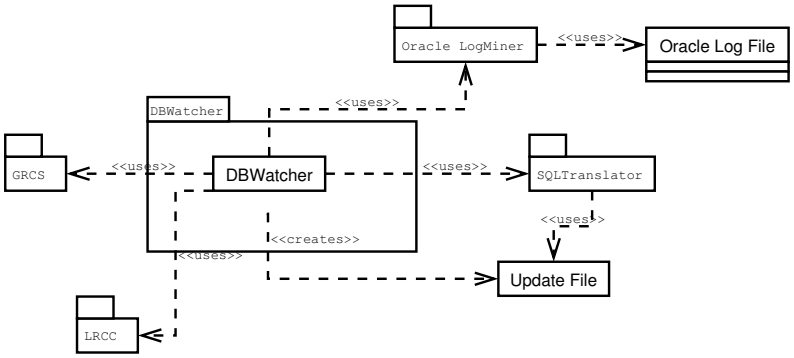

Figure 6.28: Design classes for the DBWatcher subsystem

database. The SQLTranslator module is used to translate the SQL statements retrieved with the LogMiner and modify them to make them suitable for a MySQL database. The SQLTranslator works on an update file, which is created by the DBWatcher. The DBWatcher also uses the LRCC subsystem, in order to persistently store status information, and the GRCS subsystem to trigger update propagation operations.

In Figure 6.29 the main class of this package is shown. The DBWatcher has data members to access the master database, a version number to keep track of the updates so that on the slave sites they are applied in the correct order, and time information to check the master database in the proper way, that is, without missing updates and without taking them more than once. We will see that the monitoring activity is performed by a separate thread, which runs concurrently with the main LRCS server

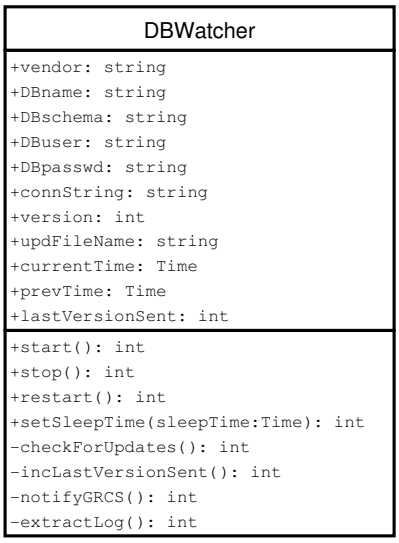

Figure 6.29: Design classes for the DBWatcher subsystem

process. Thus, the DBWatcher exposes methods to start, stop and restart a monitoring thread, and to change its polling frequency[5].

6.4.3.10 Monitoring of the Oracle Master Database

In Figure 6.30 the algorithm executed during the monitoring phase is shown. The DBWatcher class will have its own thread to execute the monitoring.

The checkForUpdates private method is used to issue a query to the LogMiner view in order to check the maximum timestamp of the statements executed since the previous check; the query will be something like:

```
select max(TIMESTAMP) from V$LOGMNR_CONTENTS
    and SEG_NAME = <list of tables to monitor>
    and timestamp > prevTime;
```

It searches in the V$LOGMNR_CONTENTS view the maximum timestamp of the statements issued on some specific monitored tables[6] (column SEG_NAME) starting from a given time.

[5]The polling frequency of the DBWatcher is set using the master LRCS configuration file, but it can be dynamically changed.

[6]This is the way it is possible to implement a partial replication, that is the replication of few selected

If the maximum timestamp is greater than the time saved in `prevTime`, then it means that the database has been updated since the last check. If not, no updates have been issued and the flow goes to the sleeping point. If updates are found, the version number of the update must be incremented. Then, the private method `extractLog` is called. It extracts the SQL statements (`SQL_REDO` column of the `V$LOGMNR_CONTENTS` view) issued on the monitored tables since the last check and puts them in a file, the so called `update unit`. The query to extract the updates will look like:

```
select SQL_REDO from V$LOGMNR_CONTENTS
    and SEG_NAME = <list of tables to monitor>
    and timestamp > prevTime;
```

At this point we have a file with statements in an Oracle-specific SQL dialect. In order to apply it to a MySQL replica, we need to use the `SQLTranslator` subsystem to translate the file, and this is done in the `translateLog` processing step.

Now the `DBWatcher` can notify the GRCS about the availability of a new update unit. If the GRCS acknowledges a successful reception of the notification, the `prevTime` value used as a bookmark for updates can be shifted to `currTime` which is the maximum timestamp of the new updates found. The `prevTime`, for recovery purposes, is also saved into the LRCC. Before reaching the sleeping point, a value holding the last version number received correctly by the GRCS is also incremented. In case the GRCS does not acknowledge having correctly received the notification, for example because of network problems, the `DBWatcher` flow simply jumps to the sleeping point, and at the next iteration will try again to send the updates previously found plus potential new ones.

tables in a database. In CONStanza, these tables are specified in the LRCS Master configuration file, see Section 6.4.3.4.

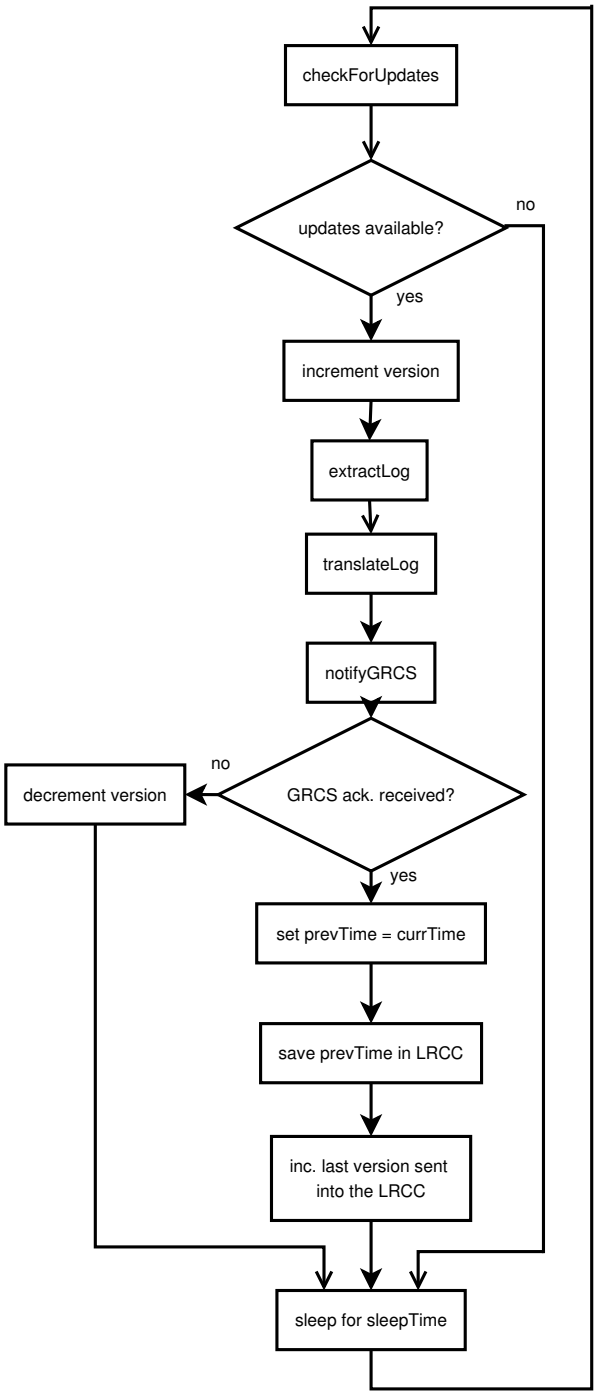

Figure 6.30: Design classes for the `DBWatcher` subsystem

6.4.3.11 SQL Translator

The `SQLTranslator` is built in an analogous way. Flex and Bison are again used to create a lexical analyser and a parser. This time the parser will read an update file containing SQL statements extracted from the master database and will create a file with SQL statements that can be applied to slave databases. In Figure 6.31 the `SQLTranslator` package is explained. In Appendix B the syntax file used to implement the Oracle to MySQL translation is shown. As we will see in the performance

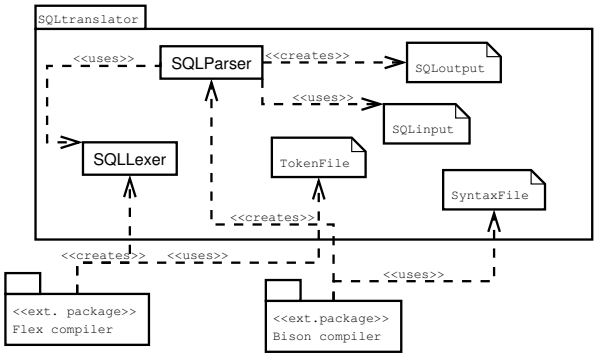

Figure 6.31: Design classes for the `SQLtranslator` subsystem

and functional tests of Chapter 7, the `SQLTranslator` syntax supports the COOL APIs for the heterogeneous replication of conditions databases.

6.4.4 CONStanza in action: Oracle to MySQL synchronisation

In this section we review all the main phases, components and interactions that are used to satisfy the Oracle to MySQL synchronisation.

As we saw in the previous sections, the CONStanza architecture is made of two main types of services: the GRCS, which is responsible for the coordination of the overall architecture, and the LRCS, which is a service deployed close to the replicas and that takes care of database monitoring (in case of a master LRCS) and update application (in case of a slave LRCS). These two servers use MySQL databases (RCC and LRCC) to store status information providing protection against network and software failures.

The CONStanza RCS software is available at the project site [2], where RPMS packages can be downloaded for source and binary installation[7]. The CONStanza

[7] At the moment only Intel 32 bit platforms with Scientific Linux CERN version 3, SLC3, and Red

User Guide explains the installation procedure, and how to configure keys and certificates for enabling the GSI. The security infrastructure, for testing purposes, can also be disabled through the GRCS and LRCS server configuration files.

Once the servers are installed, configuration files must be edited to set basic properties like database access and logging behaviour. On the master site, where an LRCS master is deployed for monitoring the Oracle master database, the `lrcs.master.conf` file allows the configuration of the access to the master database and the frequency used by the `DBWatcher` component to check the log files. It is also possible to select the tables to monitor, allowing the implementation of a basic form of partial replication at the table level. For a single master configuration like the one used in the Oracle to MySQL replication scenario, it is suggested to deploy the GRCS server at the same master site, to reduce the impact of potential network failures. At slave sites, LRCS servers are deployed and configured using the `lrcs.slave.conf` configuration file.

When the installation and configuration procedure are completed, we can start the servers in the following sequence:

1. Start the GRCS server using the `grcs-server.sh` script:

```
[rcs@oracle-db-3 etc]$ ./grcs-server.sh --help
GRCS host name is oracle-db-3.cr.cnaf.infn.it
GRCS port is 8090
GRCS debug level is 1
GRCS security is ON
grcs.conf
GRCS configuration file is: grcs.conf
Lock file is: /home/rcs/tests/t1t2tests/sbin/grcsserver.lock
Usage: ./grcs-server.sh {start|stop|restart|status}


[rcs@oracle-db-3 etc]$ ./grcs-server.sh start
GRCS host name is oracle-db-3.cr.cnaf.infn.it
GRCS port is 8090
GRCS debug level is 1
GRCS security is ON
grcs.conf
GRCS configuration file is: grcs.conf
```

Hat Enterprise Linux 3 have been fully tested.

```
Lock file is: /home/rcs/tests/t1t2tests/sbin/grcsserver.lock
Starting grcsserver
....
```

The GRCS, when started, creates the RCC database and starts listening for requests from the LRCS servers on the specified port.

2. Start the LRCS slave server using the `lrcs-server.sh` script:

```
[lrcs@pcgridtest2 etc]$ ./lrcs-server.sh start
LRCS host name is pcgridtest2.pi.infn.it
LRCS port is 8081
LRCS debug level is 1
LRCS security is ON
lrcs.slave.conf
LRCS configuration file is: lrcs.slave.conf
Lock file is: /t1t2tests/sbin/lrcsserver.lock
Starting lrcsserver
....
```

This operation must be done after the GRCS server is started, because when an LRCS server starts, one of the first things it does is to subscribe to the GRCS. Then, the LRCS slave server creates its LRCC and verifies the access to the slave replica (specified in the configuration file). If the replica is accessible, the LRCS subscribes a slave replica to the GRCS, and waits for update requests.

3. Start the LRCS master server using the `lrcs-server.sh` script:

```
[lrcs@oracle-db-3 etc]$ ./lrcs-server.sh start
LRCS host name is oracle-db-3.cr.cnaf.infn.it
LRCS port is 8091
LRCS debug level is 1
LRCS security is ON
lrcs.master.conf
LRCS configuration file is: lrcs.master.conf
Lock file is: /t1t2tests/sbin/lrcsserver.lock
Starting lrcsserver
....
```

The LRCS master also subscribes to the GRCS, creates its LRCC and creates and starts the activity of a `DBWatcher` component on the master database specified in the configuration file.

Once the service infrastructure is in place, a typical *update extraction-propagation-application* phase is described in Figure 6.32.

The `DBWatcher`, created and activated by the LRCS master, periodically uses the LogMiner in order to discover new changes on the master database. When some updates are found, they are extracted and saved in a file. This file is then used by the `SQLTranslator` to produce another file with an SQL syntax supported by MySQL. When the translation is finished, the `DBWatcher` notifies the GRCS about the availability of new updates. On the GRCS, this operation triggers the creation of a new `UpdateOperation` object in charge of the update propagation phase. Before starting the update propagation, the GRCS checks whether a quorum among the secondary replicas is reached. If so, it executes the update operation, otherwise the operation is rejected. When executing, an `UpdateOperation` object retrieves the list of all the slave LRCSs that must be contacted, and sends them an `updateDBreplica` command specifying the location of the update unit at the master site. When an LRCS does not reply to an `updateDBreplica` command the `UpdateOperation` uses the `DefOpsContainer` to save the operation so that it can be re-executed later on when the LRCS is again available. On the LRCS slave site this request triggers the creation of an `updateThread`, which retrieves the update file with a GridFTP file transfer and takes care of the update application through the `DBUpdater` component. The correct order of application of the updates is respected by the `DBUpdater`.

It is worth noting that the GridFTP file transfers used by the LRCSs to retrieve the update file happen concurrently, giving an important contribution to speeding up the update propagation phase and making it scalable with increasing numbers of slave sites. We will analyse better the performance of the RCS in the next chapter.

Another important thing to note is that the sleep time specified in the LRCS master configuration file represents only an upper limit for the polling frequency used by the `DBWatcher`. The sleep time in fact starts as soon as an `UpdateOperation` acknowledges the correct propagation of the `updateDBreplica` command.

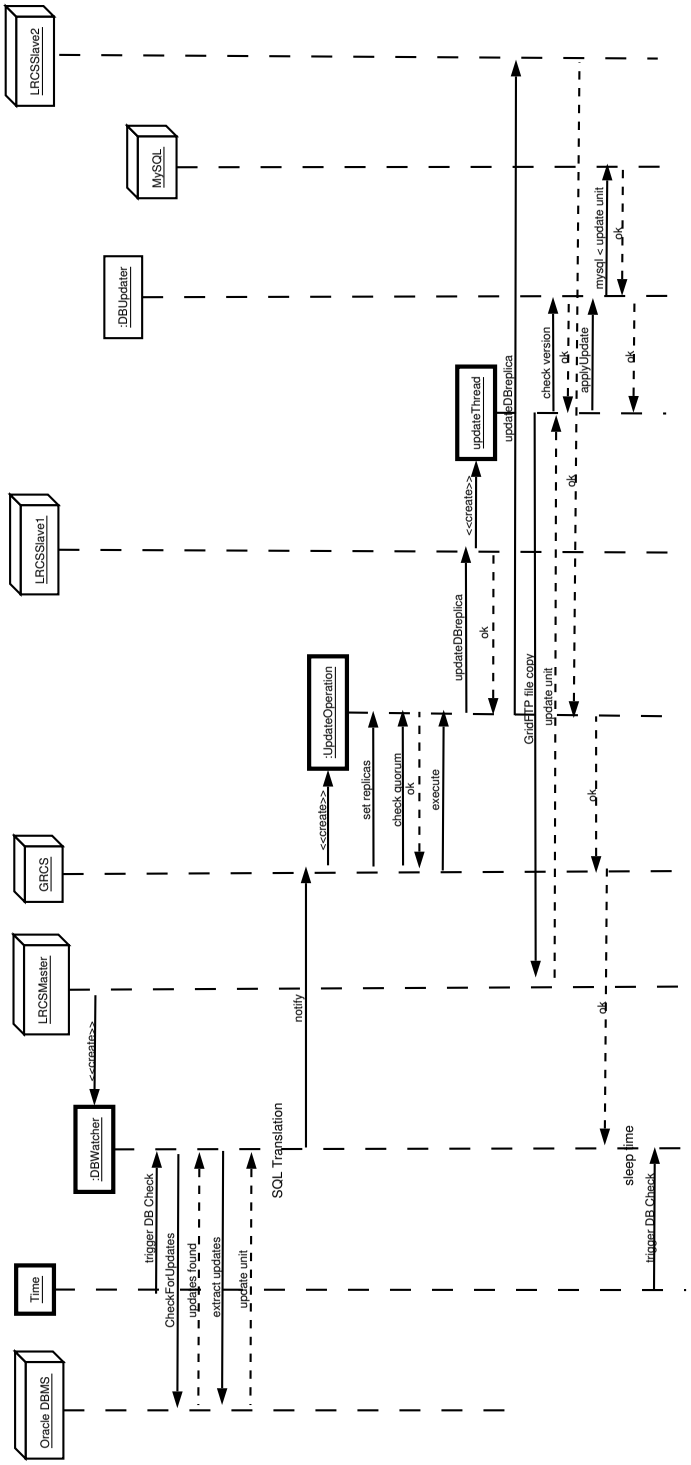

Figure 6.32: Sequence diagram of a complete Oracle to MySQL synchronisation process

6.5 CONStanza, OGSA and OGSA-DAI

The Open Grid Service Architecture (OGSA) was introduced in Section 5.3. We saw that in OGSA the middleware is made of Grid services, web services with a specific semantic and well defined interfaces in terms of lifetime management, notification and service discovery. Although the CONStanza RCS is not built as a "Grid service", as defined in OGSA, it is a web service that works in a Grid architecture and its architecture is a Service Oriented Architecture (SOA). Therefore, a re-engineering process of the CONStanza RCS to make it compliant with the OGSA standard would require a fairly short time, and would be both an interesting exercise and a valuable activity. However, during this work, we focused more on providing a reliable service that could fit the LGC/gLite middleware, that is not based on OGSA. Therefore, making the CONStanza RCS OGSA compliant has been left as a future work. Nonetheless, we expect that the OGSA standard will be used more and more in the future, thus the adaptation of the RCS to OGSA will be required.

The relation between our RCS and OGSA-DAI (see Section 5.3.1) is different. They provide complementary services: while OGSA-DAI deals with data access and integration, our RCS deals with the synchronisation of data sources. In the replication of databases, we believe in a scenario where OGSA-DAI is placed between the client and the data sources, and the RCS is placed behind the data sources, without interaction with users. This picture is shown in Figure 6.33, where heterogeneous databases are considered as data sources.

Thanks to OGSA-DAI clients could discover and access a database replicated on heterogeneous platforms kept synchronised using the CONStanza RCS.

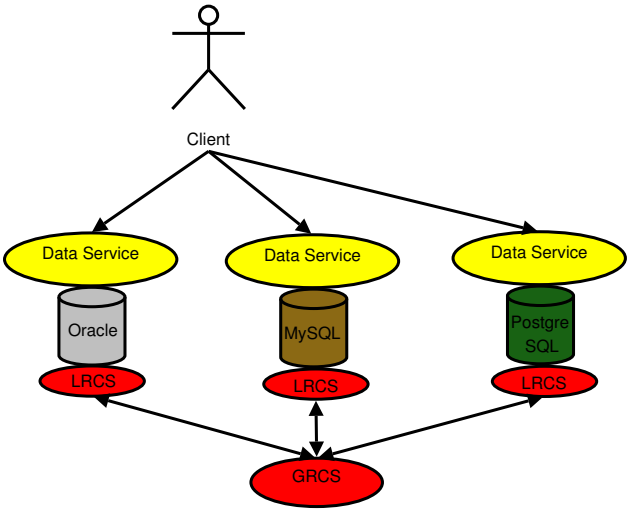

Figure 6.33: OGSA-DAI and CONStanza to manage access to consistently replicated heterogeneous databases

Chapter 7

Performance Analysis

In Chapter 6 we presented CONStanza, the Replica Consistency Service for Data Grids. We started from the requirements, functional and non-functional, then we presented the use case model and the analysis classes. We showed the RCS architecture, the GRCS and the LRCS servers, and how they collaborate to implement the use cases. We focused on two main use cases: file and database update. Being it the highest priority requirement during the development, in Section 6.4.4 we detailed the implementation of the database update use case, and in particular the synchronisation, in a single master configuration, of an Oracle database with MySQL replicas. Several tasks must be accomplished in order to realise this use case: Oracle log monitoring and log extraction, log translation, update propagation and update application at remote replicas.

In order to understand better the system, and plan optimisation tasks for the future, we analysed the performance of the RCS in that use case. The main goal of this analysis was to evaluate the system scalability, and possible system bottlenecks. The performance analysis has been done with a two-factor full factorial design model with replication [82]. This analysis allowed us to discover the most time consuming tasks, to have an idea of the responsiveness of the system and, therefore, of the lazyness of the used protocol.

The chapter is structured as follows. In Section 7.1 we present the testbed used to perform the experiments. In Section 7.2 we describe the response variables used during the experiments. In Section 7.3 instead we describe the factors, or controlled variables, and parameters, or fixed quantities. In Section 7.4 we present the model used to study the system, and specifically the effect of the factors on the response variables. In Section 7.5 we show and discuss the results, highlighting the cases

in which the model equation revealed not accurate enough to explain the observed behaviour. Finally, in Section 7.6, we summarise the conclusions of this performance analysis.

7.1 Testbed description

In Figure 7.1 the testbed used for the experiments is shown. It is made of five sites: CNAF Bologna (Italy), INFN Pisa (Italy), SNS Pisa (Italy), INFN Bari (Italy) and CERN Geneva (Switzerland) with the following hardware:

CNAF Bologna: 2 Intel(R) Xeon 2.2 GHz with 2 GB RAM.

INFN Pisa: Intel(R) Xeon 1.7 GHz with 512 MB RAM.

SNS Pisa: AMD Athlon 900 MHz with 256 MB RAM.

INFN Bari: 2 Intel(R) Xeon 3.06 GHz with 2 GB RAM.

CERN Geneva: 2 Pentium3 Coppermine 1 GHz with 512 MB RAM.

All the machines run on Scientific Linux CERN 3 apart from CERN that uses Red Hat Enterprise Linux 3, and they are all equipped with FastEthernet network cards.

7.2 Response variables

Response variables (expressed as y below) are the system variables we want to measure during the performance analysis of CONStanza. In what follows we list the seven response variables we chose, with a brief description.

1. *Response time for automatic database synchronisation*: $y_{AutUpdT}$. It can be defined as the time interval between the commit of an update operation on the master database and the time all the slave replicas have been updated. The measured time interval starts with the time recorded in the Oracle log files and finishes with the time recorded by the `DBUpdater` that prints the system time after it has successfully applied an update to the replica. Since we have more than one slave replica, we chose the maximum value printed by all the `DBUpdater` components of the testbed.

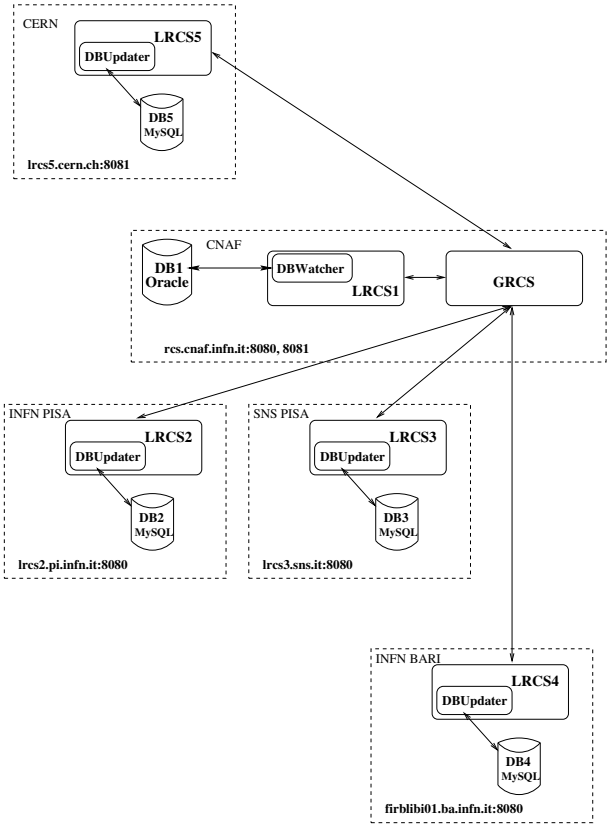

Figure 7.1: CONStanza testbed

2. *Time needed to create an update file*: $y_{LogGenT}$. This time interval is the time needed to extract the new updates from the Oracle master database using the LogMiner plus the time needed to translate these statements to make them MySQL compatible.

3. *Time needed to translate an update file*: $y_{TranslT}$. This is the time needed for the SQL translation which is also included in $y_{LogGenT}$.

4. *Time needed by the* `DBWatcher` *to notify the GRCS*: $y_{GRCSNotT}$. This time covers the main part of the update propagation process. It starts with the time the `DBWatcher` makes the call to the GRCS to request a new update operation and

ends with the time the `DBWatcher` gets a response.

5. *Time needed by the* `UpdateOperation` *object to deliver an update replica request to all the LRCSs involved*: $y_{UpdRepsT}$. This time starts with the first update replica call to the first LRCS and ends when the `UpdateOperation` object has received the response from the last LRCS; it is also included in $y_{GRCSNotT}$. It is worth stressing the fact that, as we have seen in the sequence diagram in Figure 6.32, this time does not include the actual file transfers and update of the slave databases.

6. *Time needed to retrieve the update file*: $y_{FilecpT}$. With this time we measure the time needed by LRCS slaves to retrieve the update file using GridFTP.

7. *Time needed to apply an update*: y_{DBupdT}. This is the time needed by the `DBUpdater` to apply an update file to the slave database.

The first response variable, $y_{AutUpdT}$, is the most important one since it gives an exact measurement of the laziness of the system, that is the time interval the slave replicas are in an inconsistent state with respect to the master replica. During this time, stale reads may happen. The rest of the response variables are useful to detect the most time consuming activities involved in a synchronisation process and possibly the bottlenecks of the system; the sequence diagram in Figure 7.2 highlights these time intervals to better understand their role in a synchronisation process.

All the response variables are time contributions and will be measured in milliseconds; in what follows we will refer to y_{xxxT} to indicate the response variable, and to *xxxT* to indicate the time interval that y_{xxxT} refers to. For *AutUpdT* we will also use the term “synchronisation time”.

7.3 Factors and parameters

Factors are system variables that we intentionally varied in a set of experiments to study their *effect* on response variables. Every factor has a specified number of levels that are the values they will assume during the experiments. We have chosen factors, and number and value of levels in order to analyse the system in all its main characteristics, but considering also hardware availability and time constraints. The factor list is:

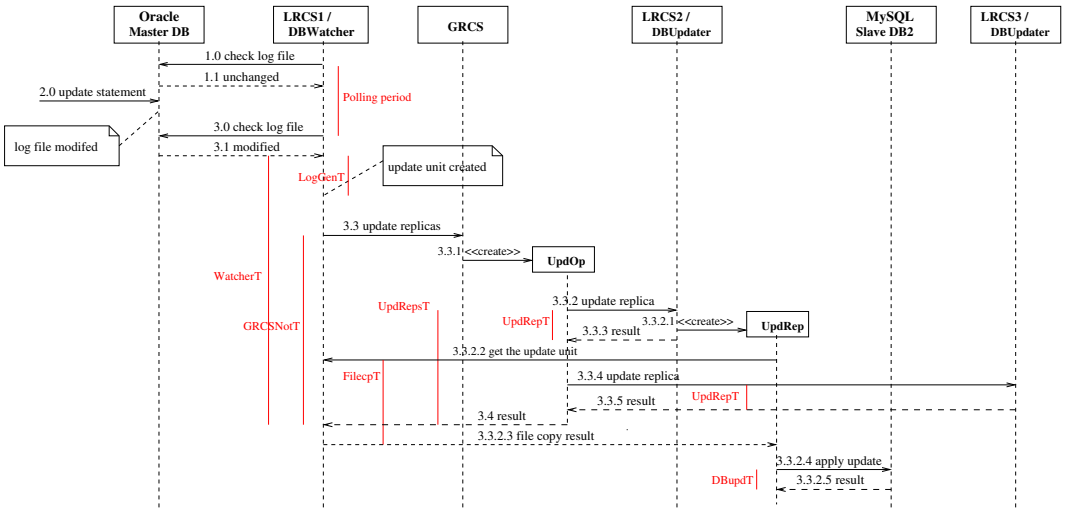

Figure 7.2: Sequence diagram for database synchronisation

- *a*: number of secondary replicas. This factor has four levels[1] (A = 4): 1,2,3,4.
- *b*: size of the update to the master in number of rows inserted. This factor has four levels (B = 4): 1, 10, 100 and 1000[2].

We decided to keep constant the following parameters which theoretically could have influenced the response variables:

- *k*: database schema: four tables of mixed number and string values.
- *l*: initial database population: empty.
- *m*: number of logical databases: 1. CONStanza is able to serve multiple logical databases, that is, with more than one master database[3]. In this case we would have had more `DBWatcher` components. Since in our main use case at the

[1] We use the letter 'A' for the number of levels of factor *a* and the letter 'B' for the number of levels of factor *b*.

[2] It is worth noting that factor *b* could be seen as a combination of other two alternative factors: the polling time of the `DBWatcher` and the frequency of the updates to the master database (see Section 7.4).

[3] Here, we do not refer to multi-master configurations but to many deployments of a single master configuration.

moment there are no requirements for managing multiple logical databases, we decided to analyse the system with only one logical database.

7.4 Experimental design

The performance study started with the selection of the response variables, that is the variables that need to be measured during the experiments. Then we analysed the system to choose factors and parameters. In our case, having two factors with four levels each, we decided to implement a *two-factor full factorial design with replication*. A single experiment is done for each combination of levels of the two factors, and every experiment is replicated R times; in our case R, also called the *replication level*, is 4. In such a design, if N is the total number of experiments needed, we obtain:

$$N = A * B * R = 4 * 4 * 4 = 64$$

An experiment consists in an execution of a script that periodically inserts some rows in the master database. The script is run each time specifying a different number of rows to be inserted (to vary factor b) and activating an increasing number of LRCS slaves at the replica sites (to vary factor a).

7.5 Experimental results and analysis

In this section we analyse the collected results and show some graphs that summarise the behaviour of the seven response variables varying the level of the factors.

Each subsection will cover the analysis of a single response variable introduced in Section 7.2. Since we will perform the same analysis for each of the seven response variables, we are going to explain it in details only for the first response variable; for the other ones we will show only the results and make the proper deductions.

An analysis will start with a simple inspection of the bar chart that summarises the results. After that, an analytical model will be used to evaluate the effects of the factors on the response variable. Finally, we will evaluate the quality of the model equation looking at how the variation of the response variable can be explained by model factors and errors.

7.5.1 Response time for automatic database synchronisation: $y_{AutUpdT}$

This is the major response variable since it gives an estimate of the laziness of the protocol. The bar chart in Figure 7.3 summarises the value of the response variable for each experiment, considering its mean value over the four replications of the experiment.

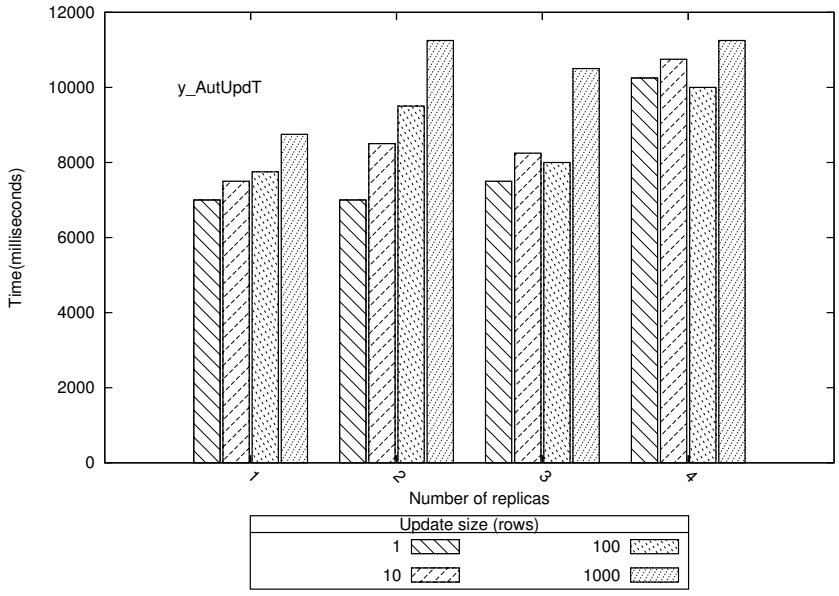

Figure 7.3: $y_{AutUpdT}$

A first deduction we can draw by looking at this graph is that, considering the levels used for each factor, the laziness of the system scales well with respect to the number of replicas. While the scalability with respect to the update size is mainly due to external components like the database management systems that control the database replicas and the file transfer protocols (GridFTP), scalability with respect to the number of replicas is highly affected by the update propagation protocol used by the RCS and, obviously, by the network latency.

7.5.1.1 Computation of effects

To analyse the results with the goal of judging the importance of each factor on the value assumed by the response variable, we use the following model:

$$y_{ijr} = \mu + \alpha_j + \beta_i + \gamma_{ij} + e_{ijr} \tag{7.1}$$

where:

- y_{ijr} is the value of the response variable of the r_{th} repetition of the experiment with factor a at level j and factor b at level i. The value y_{ijr} corresponds to a time expressed in (milli) seconds.
- μ is the overall mean response in (milli) seconds,
- α_j is the *effect* of factor a at level j,
- β_i is the *effect* of factor b at level i,
- γ_{ij} is the *effect* of the interaction between factor b and factor a when they are respectively at level i and j,
- e_{ijr} is the experimental error.

More precisely, this analysis aims at evaluating the *effect* of each factor on the deviation of the response variable from its mean value. The effects of the single factors a and b cancel out when summed over the respective levels:

$$\sum_{j=1}^{A} \alpha_j = 0; \sum_{i=1}^{B} \beta_i = 0 \tag{7.2}$$

the effects of the interactions cancel out when summed over indexes i and j:

$$\sum_{j=1}^{A} \gamma_{1j} = \sum_{j=1}^{A} \gamma_{2j} = \sum_{j=1}^{A} \gamma_{3j} = \sum_{j=1}^{A} \gamma_{4j} = \sum_{i=1}^{B} \gamma_{i1} = \sum_{i=1}^{B} \gamma_{i2} = \sum_{i=1}^{B} \gamma_{i3} = \sum_{i=1}^{B} \gamma_{i4} = 0 \tag{7.3}$$

and the errors in each experiment add up to zero[4]:

$$\sum_{r=1}^{R} e_{ijr} = 0 \tag{7.4}$$

[4]These assumptions come directly from the definition of the mean value and the model equation.

So, averaging y_{ijr} across r we obtain[5]:

$$\bar{y}_{ij.} = \mu + \alpha_j + \beta_i + \gamma_{ij} \tag{7.5}$$

and doing the same across i and j besides r :

$$\bar{y}_{i..} = \mu + \beta_i;\ \bar{y}_{.j.} = \mu + \alpha_j \tag{7.6}$$

The overall average value is:

$$\mu = \bar{y}_{...} \tag{7.7}$$

Using these results, we obtain:

$$\alpha_j = \bar{y}_{.j.} - \bar{y}_{...} \tag{7.8}$$

$$\beta_i = \bar{y}_{i..} - \bar{y}_{...} \tag{7.9}$$

$$\gamma_{ij} = \bar{y}_{ij.} - \bar{y}_{.j.} - \bar{y}_{i..} + \bar{y}_{...} \tag{7.10}$$

Effects can be computed as in Table 7.1 where we put factor a on columns and factor a on rows. Each cell contains $\bar{y}_{ij.}$ (in seconds), that is the average value of the response variable ($y_{AutUpdT}$ in this case) with factor a at level j and factor b at level i.

	1	**2**	**3**	**4**	***row mean***	β_i
1	7	7	7.5	10.25	*7.94*	*-1.05*
10	7.5	8.5	8.25	10.75	*8.75*	*-0.23*
100	7.75	9.5	8	10	*8.81*	*-0.17*
1000	8.75	11.25	10.5	11.25	*10.44*	*1.45*
col. mean	*7.75*	*9.06*	*8.56*	*10.56*	μ=*8.98*	
α_j	*-1.23*	*0.08*	*-0.42*	*1.58*		

Table 7.1: Computation of effects for $y_{AutUpdT}$

Table 7.1 says that the time needed by the service to synchronise all the replicas is, on average, 8.98 seconds. Reading the column effects, we can say that with an average number of rows inserted, the time needed with just one replicas is 1.23 seconds lower than the average one, with two replicas it is 0.08 seconds greater, with 3

[5]The dot indicates over which index or indexes the mean is taken.

replicas it is 0.42 seconds lower and with 4 replicas it is 1.58 seconds greater than the average value. The same consideration can be done reading the row effects. These values confirm what we previously said about scalability with respect to the number of replicas after looking at the bar chart (Figure 7.3), that is, the time needed to propagate the updates and synchronise all the replicas does not decrease significantly for increased levels of factor a; the difference of the maximum value and the average value of $y_{AutUpdT}$ is about 17% of the average value, while for the minimum value it is about 14% of it. Similar percentages appear also for factor b. We can state that the response time of the synchronisation protocol, in our testbed, is not critically affected by the number of the secondary replicas and by the size of the updates, when these are less than one thousands rows.

7.5.1.2 Allocation of variation

In this section we are going to figure out how the variation of the response variable from its mean value is explained by factor a, by factor b, by the interaction of these two factors and by the experimental errors. Doing this we will be able to quantify the importance of the two factors on the response variable as well as to have an idea of the quality of the model used throughout the analysis.

Squaring both sides of the model equation (Equation 7.1) and adding across all the experiments, we obtain[6]:

$$\sum_{ijr} y_{ijr}^2 = ABR\mu^2 + BR\sum_j \alpha_j^2 + AR\sum_i \beta_i^2 + R\sum_{ij} \gamma_{ij}^2 + \sum_{ijr} e_{ijr}^2 \tag{7.11}$$

The sums of squares (SS) are defined as:

$$SS_y = \sum_{ijr} y_{ijr}^2 \tag{7.12}$$

$$SS_0 = ABR\mu^2 \tag{7.13}$$

$$SS_a = BR\sum_j \alpha_j^2 \tag{7.14}$$

$$SS_b = AR\sum_i \beta_i^2 \tag{7.15}$$

[6]Recalling that cross products add to zero due to the constraint of the model (Equations 7.2, 7.3 and 7.4).

$$SS_{ab} = R\sum_{ij}\gamma_{ij}^2 \tag{7.16}$$

$$SS_e = \sum_{ijr} e_{ijr}^2 \tag{7.17}$$

Substituting the SS definitions Equation 7.11 becomes:

$$SS_y = SS_0 + SS_a + SS_b + SS_{ab} + SS_e \tag{7.18}$$

We call total variation, or total sum of squares (SS_t), the term:

$$SS_t = SS_y - SS_0 = SS_a + SS_b + SS_{ab} + SS_e \tag{7.19}$$

The total variation is made of SS_a (the variation explained by factor a), SS_b (the variation explained by factor b), SS_{ab} (the variation explained by the interaction between the two factors) and SS_e (the unexplained variation due to errors). The computations of the sums of squares are the following:

$$SS_0 = 5166.02$$
$$SS_a = 67.17$$
$$SS_b = 52.67$$
$$SS_{ab} = 16.89$$
$$SS_e = 80.25$$

From these terms we can compute the variation as percentage of the total variation:

$$explained\ by\ a = 100\frac{SS_a}{SS_t} = 30.96$$
$$explained\ by\ b = 100\frac{SS_b}{SS_t} = 24.27$$
$$explained\ by\ interaction = 100\frac{SS_{ab}}{SS_t} = 7.78$$
$$unexplained = 100\frac{SS_e}{SS_t} = 36.98$$

We can see that the percentage of variation unexplained (or explained by errors) is rather high, being 36,98% of the total variation. Consequently, for the response variable $y_{AutUpdT}$, the two-factor model should be refined to take into account other causes of variation. Furthermore, the setup of the experiment should be improved to remove any possible correlation between setup characteristics and the value of the response variable.

7.5.2 Time needed to create an update file: $y_{LogGenT}$

The meaning of this response variable has been explained in Section 7.2. In this section we will use the concepts already explained in the analysis of the response variable $y_{AutUpdT}$, thus we will not discuss the mathematical procedure. In Figure 7.4 the bar chart that summarises the results for $y_{LogGenT}$ is shown. By design, the update file creation process depends only on factor *b*, therefore we expect that the effect of factor *a* on the response variable be negligible.

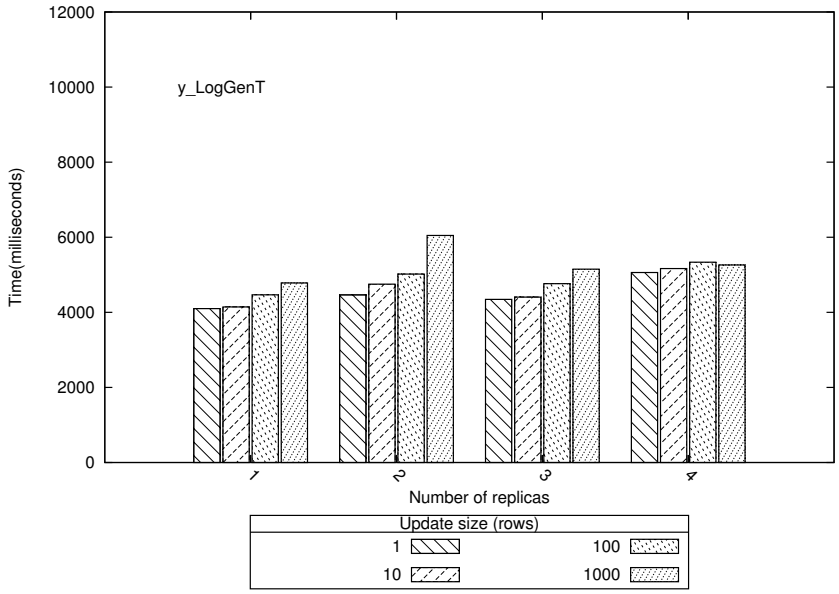

Figure 7.4: $y_{LogGenT}$

As we can see from the graph no trends are present varying the number of replicas, but just an expected trend for increasing values of the update size.

	1	**2**	**3**	**4**	***row mean***	β_i
1	4100	4462.5	4345	5060	*4491.88*	*-336.88*
10	4145	4750	4405	5165	*4616.25*	*-212.5*
100	4465	5020	4765	5335	*4896.25*	*67.5*
1000	4782.5	6047.5	5150	5262.5	*5310.63*	*481.88*
col mean	*4373.13*	*5070*	*4666.25*	*5205.63*	μ=*4828.75*	
α_j	*-455.63*	*241.25*	*-162.5*	*376.88*		

Table 7.2: Computation of effects for $y_{LogGenT}$

The table for the computation of effects is shown in Table 7.2 with values in milliseconds.

From the table we see that the average value of $y_{LogGenT}$ is about 4.8 seconds. Factor *a* should have no impact on the response variable since the update file creation phase is not influenced by the number of replicas. According to this, the column effects have no specific trend although their values are higher than one should expect (a deviation of about 10% from the mean value). Factor *b* instead shows the expected trend: with only one row inserted the update file creation time is about 300 milliseconds less than the average time while with 1000 rows the time is about 500 milliseconds greater. The time needed to generate the update file is proportional to the time needed by the LogMiner to parse the Oracle redolog files and execute the query on the LogMiner view as explained in Section 6.4.3.8. Moreover, the higher the number of updates found since the previous `DBWatcher` check, the higher the time to put these updates on the new generated file, and the larger the size of the update file. This file then must be translated (see Section 6.4.3.11), and we expect the translation time to be proportional to the update file size. We will investigate the impact of the translation time on $y_{LogGenT}$ in the next section.

With the same procedure used for $y_{AutUpdT}$, we can compute the explained variation for $y_{LogGenT}$:

$$explained\ by\ a = 28\%$$

$$explained\ by\ b = 25.5\%$$

$$explained\ by\ interaction = 9.81\%$$

$$unexplained = 36.68\%$$

The unexplained variation is high, approximately the same value as in $y_{AutUpdT}$. Variation explained by the factors are similar, but as we have already mentioned the variation explained by b, showing a clear trend, can be considered a "good" variation. The same is not true for the variation explained by a. Here the same considerations about the quality of the model equation already done for $y_{AutUpdT}$ are valid.

7.5.3 Time needed to translate an update file: $y_{TranslT}$

In Figure 7.5 the results for the translation time are summarised. Also this response variable, by design, depends only on factor b, and in fact the graph shows no differences varying the number of replicas.

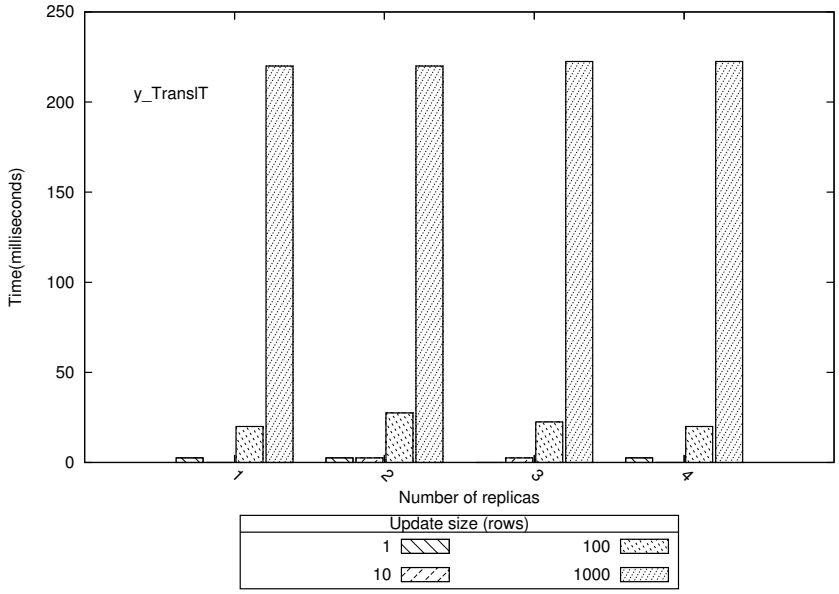

Figure 7.5: $y_{TranslT}$

From a visual inspection we can say that compared to the time needed to generate the update file ($y_{LogGenT}$), the translation time is almost negligible, being about 4% of it in the worst case. For updates less than or equal to 1000 rows, the SQL translation module is fairly efficient and is not going to be a bottleneck in the update file creation process. However, the exponential trend of $y_{TranslT}$ suggests that maybe, for higher values of factor b, $y_{TranslT}$ could play a role in increasing the value of $y_{LogGenT}$. In

	1	**2**	**3**	**4**	***row mean***	β_i
1	2.5	2.5	0	2.5	*1.88*	*-59.84*
10	0	2.5	2.5	0	*1.25*	*-60.47*
100	20	27.5	22.5	20	*22.5*	*-39.22*
1000	220	220	222.5	222.5	*221.25*	*159.53*
col. mean	*60.63*	*63.13*	*61.88*	*61.25*	*μ=61.72*	
α_j	*-1.09*	*1.41*	*0.16*	*-0.47*		

Table 7.3: Computation of effects for $y_{TranslT}$

this case, some optimisation may be required exploiting some patterns in the original SQL file, that is the input file for the SQL translation module. This aspect will be part of future work.

The table for the computation of effects is shown in Table 7.3.

According to what the graph already showed, we can say that, when the update size is less than 100 rows, the translation process is fast and its impact on the update file creation time ($y_{LogGentT}$) is negligible. When the update size reaches 1000 rows the translation time has a large increase but still being about 4% of the update file creation time. Effect of factor *a* on the response variable is negligible and does not show any trend as expected.

The computed explained variations are:

$$explained\ by\ a = 0.01\%$$

$$explained\ by\ b = 99.84\%$$

$$explained\ by\ interaction = 0.03\%$$

$$unexplained = 0.12\%$$

In this case the model covers 99.88% of the variation of the response variable, meaning that it precisely explains the behaviour of this response variable. In fact, it depends basically only on factor *b*, which is the expected result since the translation time, using a fixed set of insert statements, depends only on the number of rows inserted and is not affected by the number of replicas.

7.5.4 Time needed to notify the GRCS: $y_{GRCSNotT}$

The meaning of GRCSNotT is explained in Section 7.2. We will not analyse UpdRepsT since it is included in GRCSNotT, and does not provide further useful information. The plot for $y_{GRCSNotT}$ is shown in Figure 7.6.

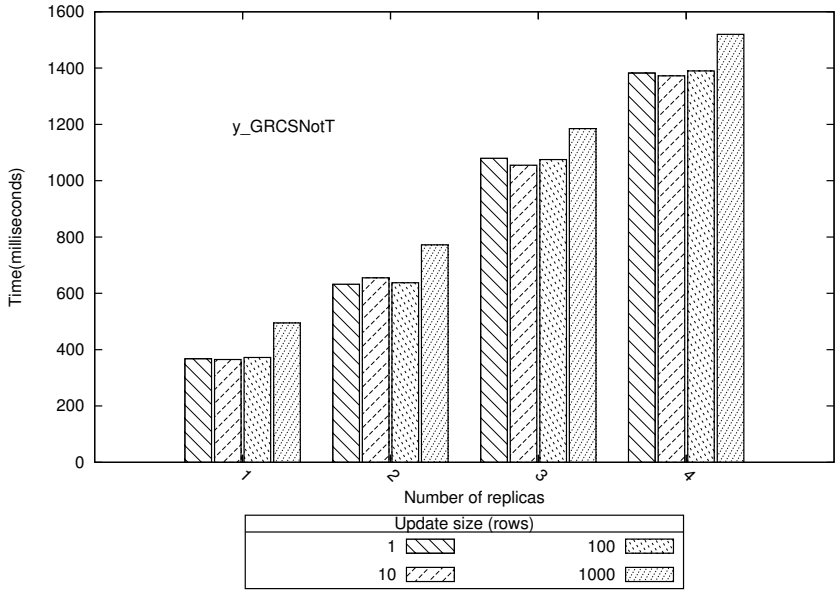

Figure 7.6: $y_{GRCSNotT}$

This response variable, by design, only depends on factor a and its trend is linear according to the implementation of this phase within the service. In fact, for each slave LRCS, the GRCS has to spawn a thread to manage the update replica request to that LRCS, so for each slave LRCS an almost constant time is added to the update propagation process. As we saw from the results of the response variable $y_{AutUpdT}$ this linear trend does not imply a linear increase of the of $y_{AutUpdT}$ with respect to factor a. In fact, as we will see, the multi-threaded implementation of the update propagation phase lets us perform the actual file transfers in a concurrent way, which has a great impact in reducing the synchronisation time. However, GRCSNotT includes a synchronous call from the DBWatcher to the GRCS (message 3.3 in the sequence diagram in Figure 7.2), that contributes to increase the overall synchronisation time.

Effects computation is shown in Table 7.4.

	1	**2**	**3**	**4**	***row mean***	β_i
1	367.5	632.5	1080	1382	*865.63*	*-31.72*
10	365	655	1055	1372	*861.88*	*-35.47*
100	372.5	637.5	1075	1390	*868.75*	*-28.59*
1000	495	772.5	1185	1520	*993.13*	*95.78*
col. mean	*400*	*674.38*	*1098.75*	*1416.25*	μ=*897.34*	
α_j	*-497.34*	*-222.97*	*201.41*	*518.91*		

Table 7.4: Computation of effects for $y_{GRCSNotT}$

The average value of $y_{GRCSNotT}$ is about 900 milliseconds. Impact of factor *b* is negligible; this is also confirmed by the allocation of variation.

$$explained\ by\ a = 97.24\%$$

$$explained\ by\ b = 1.96\%$$

$$explained\ by\ interaction = 0.04\%$$

$$unexplained = 0.76\%$$

The model suits the collected results since it explain 99.24% of of the total variation.

Comparing these results with the ones for the response variable $y_{AutUpdT}$ we can say that the time to propagate update requests ($UpdRepsT$) does not constitute a major contribution to the overall synchronisation time (*AutUpdT*).

7.5.5 Time needed to retrieve an update file: $y_{FilecpT}$

This is the time needed to make a GridFTP file transfer from the master site to the slave sites. The plot is shown in Figure 7.7. Since this time is related to a single slave site, the plot shows the value of the response variable for each slave site varying the update size (factor *b*).

Effects computation is shown in Table 7.5. The file transfer time takes from about 500 milliseconds to almost 1 second depending on the link between the master and the slave LRCS.

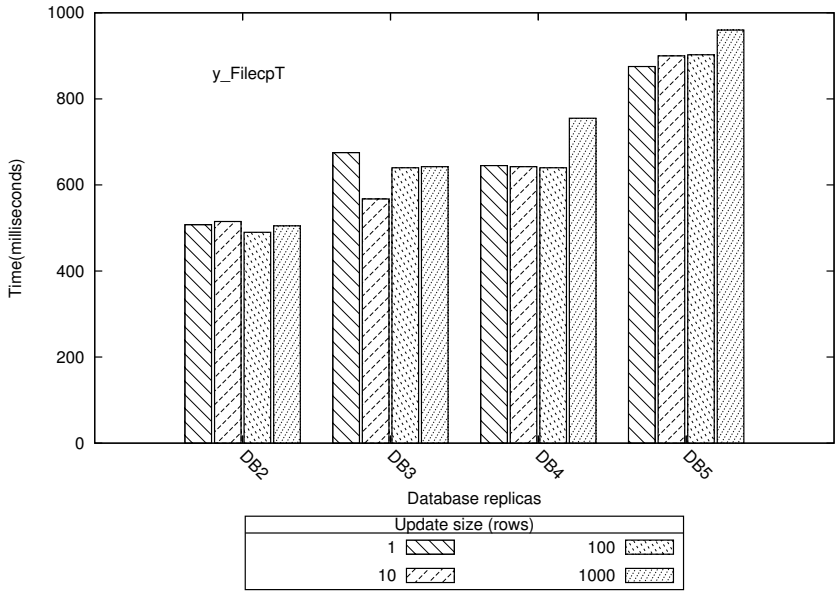

Figure 7.7: $y_{FilecpT}$

$$explained\ by\ a = 61.91\%$$

$$explained\ by\ b = 8.74\%$$

$$explained\ by\ interaction = 8.82\%$$

$$unexplained = 20.53\%$$

The average time spent for retrieving the file is 698 milliseconds. Explained variations say that, when the update size is not larger than 1000 rows, the placement of the site and network characteristics are more important than the update size as regards the update file transfer time. Also for this time interval we can say that comparing it with *AutUpdT*, it does not provide a major contribution to the overall synchronisation time, although things could change for larger numbers of rows inserted.

Possible future optimisations of this phase are related to the GridFTP server and client configurations. A SOAP based protocol for file transfer, SOAPw/Attachment [46], have been evaluated as a possible alternative to GridFTP; the tests done showed that

	DB2	DB3	DB4	DB5	*row mean*	β_i
1	507.5	675	645	875	*675.73*	*-22.03*
10	515	567.5	642.5	900	*656.25*	*-41.41*
100	490	640	640	902.5	*668.13*	*-29.53*
1000	505	942.5	755	960	*790.63*	*92.97*
col. mean	*504.38*	*706.25*	*670.63*	*909.38*	*μ=697.66*	
α_j	*-193.28*	*8.59*	*-27.03*	*211.72*		

Table 7.5: Computation of effects for $y_{FilecpT}$

	DB2	DB3	DB4	DB5	*row mean*	β_i
1	10	20	15	20	*16.25*	*-145.94*
10	20	30	20	25	*23.75*	*-138.44*
100	80	80	65	72.5	*74.38*	*-87.81*
1000	660	595	312.5	570	*534.38*	*372.19*
col. mean	*192.5*	*181.25*	*103.13*	*171.88*	*μ=162.19*	
α_j	*30.31*	*19.06*	*-59.06*	*9.69*		

Table 7.6: Computation of effects for y_{DBupdT}

SOAPw/Attachment performs better than GridFTP only when the size of the file to transfer is less than a few MBytes. The complete study can be found in [48].

7.5.6 Time needed to apply an update file: y_{DBupdT}

The plot is shown in Figure 7.8.

Also in this case the bar chart is related to the four slave sites. The behaviour of this response variable is like the one observed for $y_{TranslT}$; its contribution starts to be significant when the update size reaches one thousand rows, although not even this time, in the worst case (1000 rows), it constitutes a major contribution to the time *AutUpdT*.

Computation of effects is shown in Table 7.6. The average update time is 162.19 milliseconds. LRCS 4, that runs on the most powerful hardware platform, has the minimum time, being approximately 63% of the average value. As expected, however, the explained variations show that y_{DBupdT} mainly depends on factor *b*:

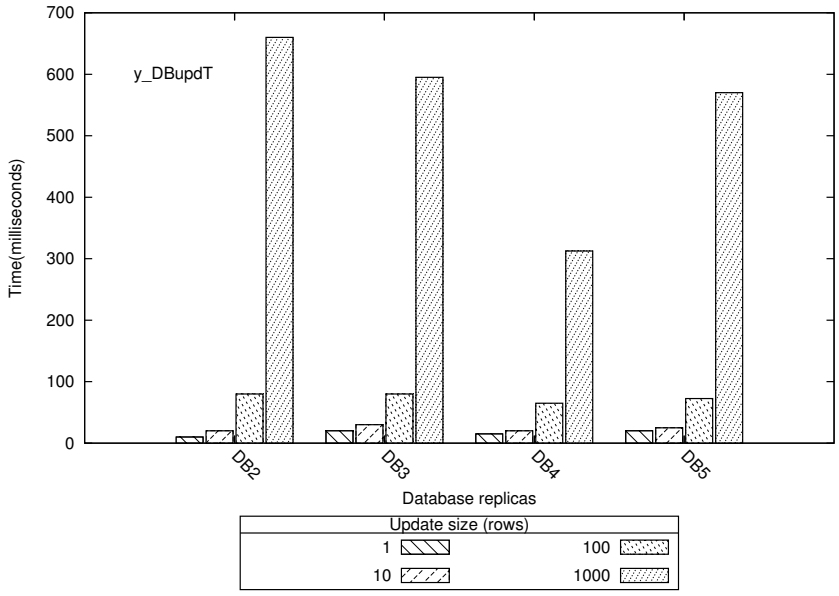

Figure 7.8: y_{DBupdT}

$$explained\ by\ a = 2.38\%$$

$$explained\ by\ b = 91.22\%$$

$$explained\ by\ interaction = 6.2\%$$

$$unexplained = 0.2\%$$

7.5.7 Sums of partial times: y_{sumsT}

To better understand how all the sub-phases we have studied so far contribute to the synchronisation time, we computed a new response variable y_{sumsT} defined as follows:

$$y_{sumsT} = y_{LogGenT} + y_{GRCSNotT} + y_{FilecpT} + y_{DBupdT}$$

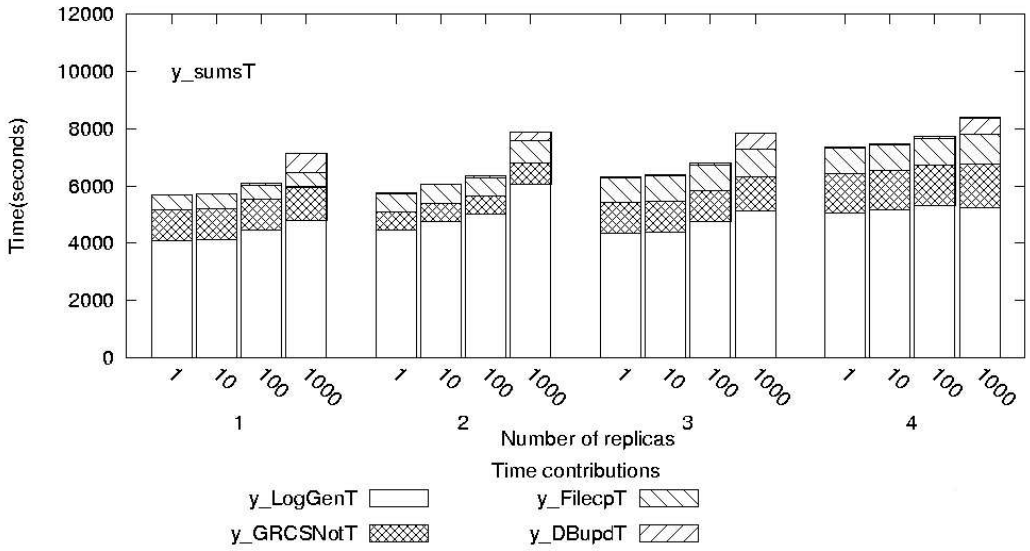

Figure 7.9: y_{sumsT}

Figure 7.9 shows the plot of y_{sumsT}.

The bar chart gives a better idea of how important the update file creation phase (*LogGenT* interval) is with respect to the other phases. Thus, the update file creation phase is the one that we will need to address to improve the performance of the service. This phase is performed by the *DBWatcher* that uses the Oracle LogMiner to discover and extract update statements issued on the master database. It involves two subsequent activations of the LogMiner which we found to be quite time consuming. Since the LogMiner can be activated and its view queried using several options, different alternatives will be investigated as part of the future work.

Comparing Figure 7.9 with Figure 7.3, we can also notice that there is a significant difference between the sum of the sub-phases times we measured and the value of the response variable $y_{AutUpdT}$, meaning that we did not take into account some time contributions. If we define:

$$y_{diffsT} = y_{AutUpdT} - y_{sumsT}$$

the plot of y_{diffsT} is shown in Figure 7.10.

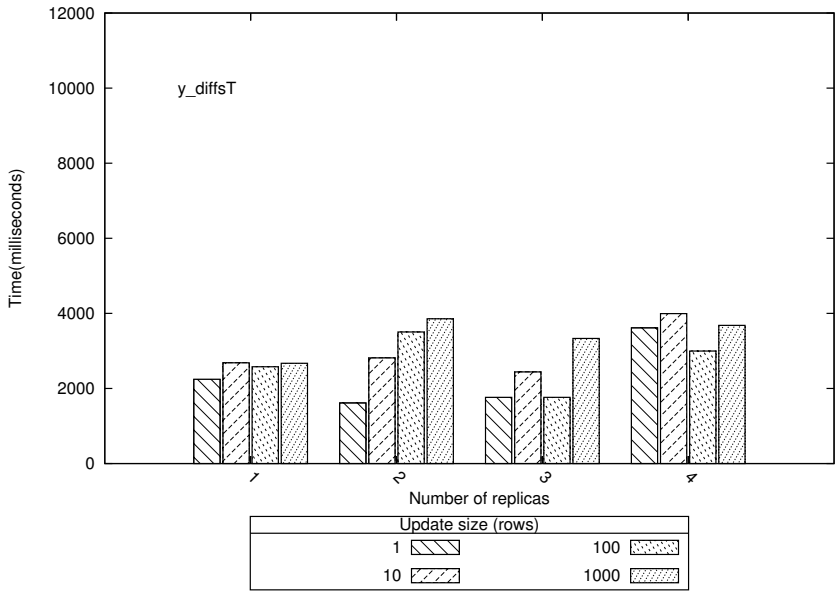

Figure 7.10: y_{diffsT}

As regards the causes of these differences ($diffsT$), we did not take into account the time needed by the `DBWatcher` to discover an update on the master database. Thus, the polling frequency is the inverse of `polltime` only when the `DBWatcher` does not find any update on the master database; otherwise, the frequency decreases by a factor that is proportional to the update file creation time ($y_{LogGenT}$), making y_{diffsT} also sensitive to factor b. Besides this, we have to consider the delay introduced by the synchronous call used by the `DBWatcher` component to notify a change to the GRCS; this call blocks until the GRCS has performed the update requests propagation phase, and the delay is proportional to the number of replica sites.

Computation of effects for y_{diffsT} is shown in Table 7.7.

The average value of y_{diffsT} is 2396.31 milliseconds and the explained variations are as follows:

$$explained\ by\ a = 27.86\%$$
$$explained\ by\ b = 32.03\%$$

	1	**2**	**3**	**4**	***row mean***	β_i
1	2015	1245	1180	2887	*1831.88*	*-564.44*
10	2455	2433	1870	3277.5	*2508.88*	*112.56*
100	2324.5	3137.5	2052.5	2262.5	*2448.75*	*52.44*
1000	2307.5	3363	2645	2867.5	*2795.75*	*399.44*
col. mean	*2280*	*2544.63*	*1936.88*	*2823.75*	μ=*2396.31*	
α_j	*-116.31*	*148.31*	*-459.44*	*427.44*		

Table 7.7: Computation of effects for y_{diffsT}

$$unexplained = 40.12\%$$

As for the first response variable analysed, $y_{AutUpdT}$, the model is not accurate enough to explain the total variation, and given what we said about the causes that generate these time differences, it is understandable that the model cannot precisely explain the behaviour of y_{diffsT}. However, comparing the explained variation for all the response variables analysed, we see that the term that is responsible for the inaccuracy of the model with regards to the synchronisation time ($y_{AutUpdT}$) is mainly the time needed for the update file creation phase, $y_{LogGenT}$.

7.6 Conclusions of Performance Analysis

In this section we collected and summarised the main observations we did throughout the previous sections of the performance study. As regards the way the different sub-phases contribute to the synchronisation time, we have seen that the main responsible is the update file creation phase (*LogGenT*). This phase is much more time consuming than the other ones and possible optimisations can be done improving the way the `DBWatcher` interacts with the LogMiner. The second largest time contribution is *GRCSNotT*, which is due to a synchronous call from the `DBWatcher` to the GRCS. An asynchronous call could decrease the value of *GRCSNotT* but it would affect the way update requests are serialized by the GRCS. This is the reason why we did not deal with it in the current release of the software. All the other sub-phases, including the update file transfer, scale well with respect to both of the factors, considering the levels used for each of them.

As regards the model used to allocate the total variation of the response variables from their mean value, we have seen that the model accurately explains the total

variations in all the cases but $y_{AutUpdT}$ and $y_{LogGenT}$. In particular, the inaccuracy of the model as regards $y_{LogGenT}$ and some time contributions we did not take into account can be considered the causes of the inaccuracy of the model for the response variable $y_{AutUpdT}$. In particular, the delay for the synchronous call from the `DBWatcher` to the GRCS (message 3.3 in Figure 7.2) and the variability of the polling frequency are not included in the model.

Chapter 8

CONStanza and Oracle Streams for Conditions Database Replication

As we have already seen in Chapter 4, one of the main use cases for CONStanza and the replication from Oracle to MySQL is the distribution of conditions databases. The four LHC experiments, and ATLAS in particular, use relational databases to save event metadata necessary for reconstruction analyses, what is generally known as conditions data. At Tier-0, conditions data are stored in Oracle, and replicated to Tier-1 sites to other Oracle databases using Oracle Streams. This replication is supported by the LCG-3D project, as we mentioned in Section 4.4. For what concerns the replication from Tier-1 sites to Tier-2 sites, more work needs to be done in order to provide a reliable solution for replicating data to open source databases.

In this chapter we show how CONStanza can be used in this scenario. CONStanza supports the synchronisation from Oracle to MySQL, and can be easily extended to support the synchronisation to other open source or commercial databases. The COOL API, introduced in Section 4.2.3, will be used to insert and retrieve data in a 3-tiers database architecture.

8.1 Testbed setup

The testbed we used for these functional tests comprises an Oracle database at CERN (Tier-0), an Oracle database at CNAF (Tier-1), and a MySQL database at INFN-Pisa

(Tier-2). The two Oracle databases have been connected and kept synchronised using Oracle Streams, while CONStanza has been deployed at CNAF (GRCS + LRCS Master) and at INFN-Pisa (LRCS Slave) for the heterogeneous synchronisation (from Oracle to MySQL). The COOL API has been used to develop two programs to insert data at the Tier-0 Oracle database and to read these data at the Tier-2 MySQL database.

8.1.1 Streams setup

For configuring Streams on the two Oracle databases we used several scripts provided by the LCG-3D project (see Section 4.4) and available at the project web page [22]. The scripts use Oracle predefined PL/SQL packages that simplify the creation and the setup of database objects; for example, most of the procedures used to setup the Capture, Propagation and Apply processes are included in a package, `DBMS_STREAMS_ADM`, that is already available in Oracle. Assuming that two Oracle databases (single instance) are running at the source and at the destination sites (Tier-0 and Tier-1), the Streams setup consists of the following steps[1]:

- Creating the Streams administrator user and grant him the necessary privileges. This operation is done both on the source and on the destination database. This administrator account will be used to create database objects in the next steps.
- Creating a database link between the source and the destination database. This link is necessary in order to propagate LCRs from a source queue to a destination queue.
- Enabling supplemental logging on the source database: this must be done for the LogMiner to work properly during the Capture activities.
- Creating the source queue and the Capture process at the source database. Streams replication is integrated with Oracle Advanced Queueing. The Capture process will use the LogMiner to extract DML and DDL changes made on the source database and will enqueue these changes as LCRs into the source queue.
- Creating the Propagation process at the source database. The Propagation process is the process in charge of propagating LCRs from the source queue to the destination queue. When creating the Propagation process we have to specify the database link created in the second step.

[1] We outline the main phases without giving technical details; these can be found in [22].

- Creating the destination queue and the Apply process at the destination database. The Apply process is the process that takes LCRs from the destination queue and applies them to the destination database.

- Test the infrastructure. After the previous steps, we tested the infrastructure by issuing SQL DDL and DML queries on the source database and verifying the correct application at the destination site. A simple test would be that of creating a table with some records on the source database and querying these records at the destination database.

These steps have been successfully done. After preparing the Streams infrastructure we passed to the installation and configuration of the CONStanza servers.

8.1.2 CONStanza setup

At the Tier-1 site, the destination database for Oracle Streams has been used as master database for CONStanza. Here, a GRCS server and an LRCS server have been installed and configured. After preparing the security infrastructure, installing keys and certificates, we started configuring the servers. In the GRCS server configuration file (refer to the Appendix A for the GRCS configuration file options) we specified a logical database named "LDBCOOL"; when the GRCS server starts, this logical database is automatically subscribed, initially without replicas.

For configuring the master LRCS server, the file `lrcs.master.conf` has been edited, providing all the necessary options including the details about a database replica named "COOLDB1". When the master LRCS server starts, after subscribing itself to the GRCS, it automatically subscribes the database replica "COOLDB1" as replica of the logical database "LDBCOOL".

Once the configuration files have been edited, we started the GRCS server. Before starting the LRCS server, we configured the slave LRCS server at the Tier-2 site. Here, an empty MySQL database named "COOLDB2" was created; this replica will be automatically subscribed as a replica of the logical database "LDBCOOL" as soon as the slave LRCS server starts.

Then, we started the slave LRCS server and, after, the master LRCS server. At this time, all the synchronisation infrastructure is in place, and every change to the Oracle database at Tier-0 will be propagated and applied, first at Tier-1, using Oracle Streams, and then at Tier-2, using CONStanza. The scenario is depicted in Figure 8.1 The thick lines show the data flow, while the thin ones among the RCS servers show communications needed for implementing the update propagation.

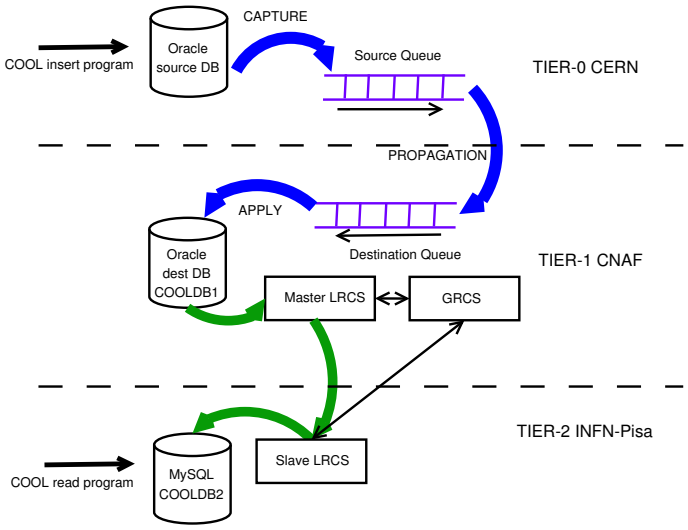

Figure 8.1: Scenario for the replication of conditions databases using Oracle Streams and CONStanza

8.2 Testing COOL insertions and retrievals

The functional test consists in inserting conditions data into the database at Tier-0 (Oracle) and reading these data at the Tier-2 replica (MySQL). More specifically, we insert and retrieve data into/from COOL SVSC folders (see Section 4.2.3), using fixed intervals of one second as IOV and integer numbers as payload.

To insert data we developed two scripts:

- `t1t2insert`: to create a new COOL database with a SVSC folder and insert conditions data in this folder. The script takes two arguments:

  ```
  $./t1t2insert.csh
  Usage: t1t2insert.csh nfolder nobjects
  ```

  The first one is a number used to create the folder name; for example when *nfolder* is 1, the folder named *folder1* is created and *nobject* objects are created in it, with IOV [0.1), [1,2), [2,3) and so on. The payload that is inserted into

the objects is an integer number equal to "*nfolder* + 10 + *until*"[2], so that each object takes a different payload.

- `t1t2insert-nocreate`: to insert objects into an existing folder, appending them on top of the existing ones. This script takes three arguments:

  ```
  $./t1t2insert-nocreate
  Usage: t1t2insert-nocreate.csh nfolder nobjects offset
  ```

  The first two arguments are the same as in the previous script, and are used to choose the folder to work on and to state how many objects will be inserted. Finally, the last argument takes an offset to avoid IOV overlapping with existing objects.

A typical output of the execution of the two program is:

```
$./t1t2insert.csh 1 2
nfolder = 1
nobjects = 2
Start
Creating a new COOL database
Creating folder /folder1
Inserting 2 objects into folder /folder1
End

$./t1t2insert-nocreate.csh 1 3 2
nfolder = 1
nobjects = 3
offset = 3
Start
Opening an existing COOL database, retrieving folder /folder1
Inserting 3 objects into folder /folder1
End
```

To retrieve data we used a third script:

```
$./t1t2retrieval.csh
Usage: t1t2retrieval.csh nfolder since until
```

[2]We recall that the *until* value is the upper bound of the IOV.

that takes three arguments. The first one is used to choose the folder to retrieve data from. The second and the third one are used to specify the interval of validity where objects will be retrieved.

After opening a COOL database, the program retrieves the folder and then all the objects in the specified IOV. To check that the right number of objects is retrieved, we inserted IOVs with a fixed and known length. This check is done inside the retrieval script. After this check, the program also checks the payload of all the objects retrieved. The payload is a function of the folder number and the object IOV; in this way the retrieval program can check whether each payload contains the right number.

A typical execution of the retrieval script is:

```
$t1t2retrieval.csh 1 0 5
nfolder = 1
since = 0
until =5
COOL DB opened
Start reading folder /folder1
Ok: found correct number of objects in IOV
Ok: payload is: 12
Ok: payload is: 13
Ok: payload is: 14
Ok: payload is: 15
Ok: payload is: 16
Data retrieval finished with 0 errors
```

The functional test completed successfully. This demonstrates that, from a functional point of view, CONStanza can be used, together with Oracle Streams, for synchronising conditions data between an Oracle database at Tier-0 and a MySQL database at Tier-2.

Chapter 9

Conclusions and Future Work

This thesis presented a study of the replica consistency problem in Grid computing and proposed a Replica Consistency Service to synchronise both files and heterogeneous databases in a Grid environment.

Starting from the basic theory of concurrency control in centralised and distributed databases, we studied different synchronisation techniques used to enforce replica consistency in wide area replicated data stores. We focused on optimistic replication, that synchronises replicas in a relaxed way, and we saw different design choices that can be used.

Since the main application field of our service was within the Grid community, and in particular Grid applications in High Energy Physics (HEP), we presented this domain and the Grid middleware solutions currently in use. These solutions, while providing full support for data replication, lack a replica synchronisation system that allows users to modify a replica without introducing inconsistency with respect to the other replicas. We emphasised that both files and databases need to be synchronised in a Grid. In particular, for database synchronisation, a large class of applications require the synchronisation of heterogeneous (different vendors) databases.

The main issues in providing such a synchronisation service, like scalability to the number of replicas, security, site unavailability and resource heterogeneity, have been discussed. Then, we presented the CONStanza project, and the Replica Consistency Service we developed during these Ph.D. studies. The design of the service has been thoroughly described, from the requirements collection to the implementation. We focused on the synchronisation of heterogeneous databases, and in particular on the synchronisation of an Oracle master database with many MySQL slave replicas. A single master approach has been chosen in order to provide a simple but stable

solution that satisfies critical requirements like security, fault tolerance and smooth installation, configuration and usage.

In CONStanza, the Oracle master database log files are checked with a configurable periodicity; changes are extracted, propagated (with a GridFTP file transfer), and applied to remote MySQL replicas. The problem of the different SQL dialects used by the two database vendors has been solved by building an SQL translation module which performs specific data transformations, on-the-fly, before propagating the updates.

A detailed performance analysis showed that, in our testbed, an average of 10 seconds are needed to propagate and apply updates (up to 1000 rows) to four slave replicas. We also found out that the most time consuming phase in the log extraction, transformation, propagation and application process was the monitoring of the master database, which relies on the Oracle LogMiner utility. The SQL translation phase demonstrated to be efficient, with a negligible impact on the overall system. The propagation of the updates scales well with increasing the number of replicas thanks to the multithreaded implementation of the file transfer procedure.

Then, using the COOL API, we tested CONStanza in a real use case, where a conditions database was replicated and kept synchronised in three tiers architecture. The replication from an Oracle database at Tier-0 (CERN) to another Oracle database at Tier-1 (INFN-CNAF) was done using Oracle Streams, while CONStanza was used to replicate the database at Tier-2 (INFN-Pisa), in a MySQL database. The overall replication system successfully replicated the conditions database. This use case is of particular interest to HEP experiments like ATLAS, where COOL is used for storing and retrieving conditions data.

9.1 Future Work

In this thesis, we described the architecture and the implementation of a Replica Consistency Service, focusing on a specific use case for heterogeneous database replication. Many interesting works could follow and enhance the CONStanza RCS; we describe below some of the ones that could be of more interest to the Grid community.

9.1.1 Application to Biomedical databases

File based databases are used in biomedical research to store data for remote, distributed analyses. We believe that an interesting work could be done to enhance

CONStanza to support the replication of biomedical file based databases. In this case, a database is a collection of files within a directory structure. In particular, the database monitoring phase in CONStanza should be enhanced to provide an alternative implementation of the `DBWatcher` component, to able to monitor files in a given directory tree. Database updates in this case are: creation of new files, modification of existing ones, and deletion of files. These updates should be monitored by the new `DBWatcher` and propagated to remote sites with minimal changes on the current CONStanza architecture. This use case has been discussed in [42].

9.1.2 Multi-master protocols

Multi-master protocols have not been used in our service implementation. The main reason of this was the lack of clear requirements from our application fields. Multi-master protocols can be designed only with clear requirements on how to resolve conflicts, and the scalability for increasing numbers of master replicas must be carefully analysed. In CONStanza, a multimaster protocol can be implemented enhancing the capabilities of the LRCS servers. In case of database synchronisation, both a `DBWatcher` and `DBUpdater` should be deployed at each master database, and a conflict resolution procedure should be implemented on the `DBUpdater`.

9.1.3 Full support of the COOL API

Currently the full support of the COOL API must be tested: the functional test in Chapter 8 was performed using a COOL Single-Version Single-Channel (SVSC) folder. Other COOL scenarios (see Section 4.2.3) must be tested.

9.1.4 An OGSA implementation of the RCS

In Section 5.3, we reviewed the Open Grid Service Architecture (OGSA). Grid services compliant with the OGSA model are becoming widely used and are implemented in the Globus Toolkit. As mentioned in Section 6.5, a re-engineering process of the CONStanza RCS to make it OGSA compliant would be both an interesting exercise and a useful contribution to the Grid community. The services offered by the RCS could be integrated with OGSA-DAI and OGSA-DQP to provide full support for database access and management in Grid computing.

Chapter 10

Acknoweledgements

The acknowledgements chapter is the chapter I like best in this thesis, because it gives me the opportunity to mention many people that helped and supported me during my doctoral studies. First of all, I would like to thank my supervisors, and especially Andrea, who, with his experience, wisdom and friendship, accompanied me throughout my Ph.D. studies and contributed not only to model my way of doing research but also my personality. I really enjoyed collaborating with him and I am very grateful for all the time and effort he put in this work. A big thanks goes to Flavia, my "hidden" tutor and every-day adviser. I am very grateful to her for giving me the opportunity to live this experience, from Tsukuba to Geneva. To close the CONStanza circle, I would also like to thank Heinz, for his collaboration and his contribution to the final review of the thesis. Thank you to all of them for the effort they put in the CONStanza project, and for their friendship.

For the time spent at CERN as doctoral student, I would like to thank my CERN supervisor Dirk, for making this experience at CERN the most interesting and enjoying of my Ph.D. career. With him, I would also like to thank all the other members of the group, and especially Andrea and Romain for their collaboration within the COOL project. Thanks to Eva for her help with Streams and LogMiner issues, and to the DBAs Miguel, Jacek, Luca and David for their kind and professional availability. At CERN I had the possibility to work with very pleasant and professional people, and all of them contributed to make my time spent at CERN unique.

I would also like to thank the Department of Information Engineering at the University of Pisa, and the Italian National Institute for Nuclear Physics (INFN) in Pisa and at CNAF, where this work has been done.

Thanks to Karolin, for her love. She helped me with the writing and the reviewing of the thesis, and encouraged me during all this time in Geneva. I am indebted to her for all the time and energy she spent to support this big effort.

And thanks to all my family for their support, and especially to my parents, the main artificers of my university career; none of this would have happened without their sacrifices.

Appendix A

Configuration files

A.1 GRCS

An example of profile script for a GRCS server is the following:

```
#!/bin/bash
#
# description: GRCS server profile script

GRCS_HOST="oraclex.cr.cnaf.infn.it"
GRCS_PORT=8090
GRCS_DEBUG="1"
GRCS_GSI="ON"
GRCS_CONF="grcs.conf"
```

An example of configuration file for a GRCS is the following:

```
#This is an example of configuration file for a GRCS server

[GRCS]
# here we put GRCS properties
#name = the name used to identify an GRCS
name = GRCS
#endpoint = host and port used by the server
```

```
endpoint = oraclex.cr.cnaf.infn.it:8090
#protocol = protocol used for update propagation (single-master)
protocol = ASYNCH_ONE_MASTER
#quorum = quorum used for update propagation
quorum = 1
#lids = list of logical dataset to subscribe
lids = (LDB1)

[LOGGING]
# here we put information to drive log message generation
#filename = name where output messages are collected
filename = GRCSout.txt
#dirname = directory where output messages are stored
dirname = /tmp
#sizelimit = limit the size of output files in KB (0 = no limits)
sizelimit = 0
#filelevel = level of output messages on file
#(FATAL,ERROR,INFO,DEBUG)
filelevel = RCS_DEBUG
#consolelevel = level of output messages on console
#(FATAL,ERROR,INFO,DEBUG)
consolelevel = RCS_INFO

[RCC]
# these are information related to the GRCS catalogue
host = localhost
bckphost = pcgridtest3.pi.infn.it
port = 3306
dbname = RCC
username = rcs
password = constanza
```

A.2 LRCS

An example of profile script for the LRCS server is the following:

```
#!/bin/bash
```

```
#
# description: LRCS server profile script

LRCS_HOST="oraclex.cr.cnaf.infn.it"
LRCS_PORT=8091
LRCS_DEBUG="1"
LRCS_GSI="ON"
LRCS_CONF="lrcs.master.conf"
```

An example of configuration file for a slave LRCS is the following:

```
[LRCS]
# here we put LRCS properties
#name = the name used to identify the LRCS
name = LRCS2
#endpoint = host and port used by the LRCS server
endpoint = pcgrid3.pi.infn.it:8081
#grcsendpoint = host and port used by the GRCS server
grcsendpoint = pcgrid2.pi.infn.it:8080
#updatesdir = the default location for incoming updates when
#an LRCS acts as slave
updatesdir = /tmp/updates
#logsdir = the default location for log files extracted from
#the master replica when the LRCS acts as master
logsdir = /tmp/logs

[LOGGING]
# here we put information to drive log message generation
#filename = name where output messages are collected
filename = LRCSout.txt
#dirname = directory where output messages are stored
dirname = /var/log/lrcs
#sizelimit = limit the size of output files in KB (0 = no limits)
sizelimit = 0
#filelevel = level of output messages on file
#(FATAL,ERROR,INFO,DEBUG)
filelevel = RCS_DEBUG
```

```
#consolelevel = level of output messages on console
#(FATAL,ERROR,INFO,DEBUG)
consolelevel = RCS_INFO

[LRCC]
# these are information related to the LRCS database (LRCC)
host = localhost
port = 3306
dbname = LRCC
username = rcs
password = constanza

#then there are sections for database replicas.
#
[DB2]
#repname = database name
repname = DB2
#schema = schema name. Relevant only for Oracle DB
schema = none
#repuser = db replica user name
repuser = rcs
#reppassw = db replica password
reppassw = constanza
#connstring = ORACLE connection string
#(//host:port/service_name). Relevant only for Oracle DB
connstring = none
#vendor = ORACLE or MYSQL
vendor = MYSQL
#master = yes for master replicas, no otherwise
master = no
#LId = logical identifier of that replica
LId = LDB
#logfile = log file names, comma separated and between
#brackets. Relevant only #for Master DB
logfiles = none
#logsdir = location where the log files will be stored
#(none for default location indicated in the LRCS section)
```

```
logsdir = none
#pooltime = interval between to successive log file check.
#Relevant only for Master DB
pooltime = none
#tables = list of tables to monitor, comma separated and
#between brackets. Relevant only for Master DB
tables = none
```

An example of configuration file for a master LRCS is the following:

```
#This is an example of configuration file
#for an LRCS acting as Master

[LRCS]
# here we put LRCS properties
#name = the name used to identify the LRCS
name = LRCS1
#endpoint = host and port used by the LRCS server
endpoint = oraclex.cr.cnaf.infn.it:8091
#grcsendpoint = host and port used by the GRCS server
grcsendpoint = oraclex.cr.cnaf.infn.it:8090
#updatesdir = the default location for incoming updates when
#an LRCS acts as slave
updatesdir = /tmp/updates
#logsdir = the default location for log files extracted from
#the master replica when the LRCS acts as master
logsdir = /tmp/logs

[LOGGING]
# here we put information to drive log message generation
#filename = name where output messages are collected
filename = LRCSout.txt
#dirname = directory where output messages are stored
dirname = /tmp
#sizelimit = limit the size of output files in KB (0 = no limits)
sizelimit = 0
#filelevel = level of output messages on file
```

```
#(FATAL,ERROR,INFO,DEBUG)
filelevel = RCS_DEBUG
#consolelevel = level of output messages on console
#(FATAL,ERROR,INFO,DEBUG)
consolelevel = RCS_INFO

[LRCC]
# these are information related to the LRCS database (LRCC)
host = localhost
port = 3306
dbname = LRCC
username = rcs
password = constanza

#then there are sections for database replicas.
#
[COOLDB1]
#repname = database name
repname = COOLDB1
#schema = schema name
schema = GIANNI
#repuser = user name
repuser = GIANNI
#reppassw = password
reppassw = gianni
#connstring = ORACLE connection string (//host:port/service_name)
connstring = //oraclex.cnaf.infn.it:1521/pisatest.cr.cnaf.infn.it
#vendor = ORACLE or MYSQL
vendor = ORACLE
#master = yes for master replicas, no otherwise
master = yes
#LId = logical identifier of that replica
LId = LDB1
#logfile = log file names, comma separated and between brackets
logfiles = none
#logsdir = location where the log files will be stored (none for
#default location indicated in the LRCS section)
```

```
logsdir = none
#pooltime = interval between to successive log file check
pooltime = 10
#tables = list of tables to monitor, comma separated and
#between brackets
tables = (NONE)
```

Appendix B

SQL Translator syntax

```
input ::= lines

lines ::= { line | lines line }

line ::= { insert ; | delete ; | update ; }

insert ::= INSERT single_table_insert

single_table_insert ::= insert_into_clause values_clause

insert_into_clause ::= INTO dml_table_expression_clause
                       [ (column [, column ]...) ]

values_clause ::= VALUES ( { expr | DEFAULT }
                     [, { expr | DEFAULT } ]... )

dml_table_expression_clause ::= schema . table

delete ::= DELETE [ FROM ] dml_table_expression_clause i
                             [ where_clause ]

where_clause ::= WHERE condition

condition ::= { simple_comparison_condition
```

```
                | range_condition
                | null_condition
                | compound_condition
              }

simple_comparison_condition ::= expr COMP expr

range_condition ::= expr [ NOT ] BETWEEN expr AND expr

null_condition ::= expr IS [ NOT ] NULL

compound_condition ::=
                { ( condition )
                 | NOT condition
                 | { simple_comparison_condition { AND | OR }
                     simple_comparison_condition
                     [ { simple_comparison_condition { AND | OR }
                         simple_comparison_condition
                       }
                     ]...
                   }
                }

update ::= UPDATE dml_table_expression_clause update_set_clause
           [ where_clause ]

update_set_clause :== SET column ASS { expr | DEFAULT}
                      [ , column ASS { expr | DEFAULT} ]...

expr ::= { simple_expression
           | compound_expression
           | function_expressio
         }

simple_expression ::= { [ dml_table_expression_clause . ]
                            { column | ROWID }
                        | TEXT
```

```
                        | NUMBER
                        | NULL
                      }

compound_expression ::= { ( expr )
                        | { ADD | SUB } expr
                        | expr { MUL | DIV | ADD | SUB } expr
                      }

function_expression ::= single_row_function

single_row_function ::= { numeric_function
                        | datetime_function
                        | conversion_function
                        | miscellaneous_single_row_function
                      }

numeric_function ::= { ABS ( NUMBER )
                     | EXP ( NUMBER )
                     | LN ( NUMBER )
                     | ROUND ( NUMBER [ , integer ]
                     | SIGN ( NUMBER )
                     | SQRT ( NUMBER )
                    }

datetime_function ::= { CURRENT_DATE
                      | CURRENT_TIMESTAMP [ ( NUMBER ) ]
                      | SYSDATE
                      | SYSTIMESTAMP
                     }

conversion_function ::= { SCN_TO_TIMESTAMP ( TEXT )
                        | TIMESTAMP_TO_SCN ( TEXT )
                        | HEXTORAW ( TEXT )
                      }

miscellaneous_single_row_function ::= { { EMPTY_BLOB | EMPTY_CLOB }
```

```
                                        | UID
                                        | USER
                                      }

integer ::= NUMBER

schema ::= STC_NAME

table ::= STC_NAME

column ::= STC_NAME
```

Appendix C

LogMiner Utility Package

In this appendice we show the PL/SQL package used to start and test the LogMiner.

```
/*
Package that collects some LogMiner testing utilities
*/

set serveroutput on;

drop table CONSTABLE;

create table CONSTABLE (
    name varchar2(20),
    age number(2)
);

--Package Declaration
create or replace
package LogMinerUtils_pkg as

/*
Print info on redo groups, files, threads etc..
*/
procedure printRedoInfo;

/*
```

```
Start the LogMiner with start and end time plus continuous mine option.
*/
procedure startLogMinerCMine(t1 varchar2, t2 varchar2);

/*
Start the LogMiner without using the continuous mine option
and specifying all the online redo logs.
*/
procedure startLogMiner;

/*
Query the LogMiner view in order to collect updates done in [t1,t2)
issued by 'user'. Return number of statements found in stmtfound.
*/
procedure queryLogMiner(t1 varchar2, t2 varchar2, user varchar2,
                        stmtfound out number);

/*
Terminate the LogMiner
*/
procedure closeLogMiner;

/*
Insert a row into the test table CONSTABLE.
*/
procedure insertIntoTestTable;

/*
Insert a row and retrieved it with the LogMiner.
Use the continuous mine option.
stmt found returns the number of stmts found and elapsed time the
time needed to start and query the LogMiner.
Start the LogMiner with t1,t2
*/
procedure test1(elapsedTime out number, stmtfound out number,
                user varchar2);
```

```
/* Same as test1 but starting the LogMiner without cmine, adding all
the redolog files
*/
procedure test2(elapsedTime out number, stmtfound out number,
                user varchar2);

/*
Execute 'times' times the test1 procedure and print the average
elapsed time to execute the query to the Logminer.
When the query does not secceed in finding the statement, it exits
the loop and print the iteration where the error was found.
Each loop run with a 3 sec pause.
This procedure is made in order to discover non deterministic errors.
*/
procedure test1n(user varchar2, times number);

/* Same as test1n but using test2
*/
procedure test2n(user varchar2, times number);

LogMinerUtils_pkg;


--Package Definition
create or replace
package body LogMinerUtils_pkg as

procedure printRedoInfo is
begin
    --Print number of Redo Groups
    for redogroups in ( select count(distinct group#) numero
                          from v$log
                      )
    loop
        dbms_output.PUT_LINE('Number of redo groups: '
                                || to_char(redogroups.numero));
    end loop;
```

```
    --Print size of Redo Groups
    for sizes in ( select distinct(bytes) numero
                     from v$log
                  )
    loop
        dbms_output.PUT_LINE('Size in Bytes: ' || to_char(sizes.numero));
    end loop;

    --Print number of Threads
    for thread in (SELECT count(distinct(thread#)) numero
                     FROM v$log
                   )
    loop
        dbms_output.put_line('Number of threads: ' || thread.numero);
    end loop;

    --Print Log Files
    for member in (select member name from v$logfile)
    loop
        dbms_output.put_line('Log file: ' || member.name);
    end loop;

end printRedoInfo;

procedure startLogMinerCMine(t1 varchar2, t2 varchar2) is
begin
    dbms_output.PUT_LINE('Starting LogMiner: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

    sys.dbms_logmnr.start_logmnr(options =>
            sys.dbms_logmnr.dict_from_online_catalog
            + sys.dbms_logmnr.committed_data_only
            + sys.dbms_logmnr.no_rowid_in_stmt
            + sys.dbms_logmnr.continuous_mine
            , starttime => to_date(t1,'DD-MON-YYYY HH24:MI:SS')
            , endtime => to_date(t2,'DD-MON-YYYY HH24:MI:SS'));
```

```
    dbms_output.PUT_LINE('LogMiner started: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));
end startLogMinerCMine;

procedure startLogMiner is
begin
    --add all the redo log files to the LogMiner
    for logfile in (select  member name
                      from v$logfile
                    ) loop
        begin
            sys.dbms_logmnr.remove_logfile(logfile.name);
        exception
            when OTHERS then
                dbms_output.put_line('ORA-01290');
        end;
        sys.dbms_logmnr.add_logfile(logfile.name);
        dbms_output.put_line('logfile: ' || logfile.name || ' added');
    end loop;

    dbms_output.PUT_LINE('Starting LogMiner: ' ||
                      to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

    sys.dbms_logmnr.start_logmnr(options =>
            sys.dbms_logmnr.dict_from_online_catalog
            + sys.dbms_logmnr.committed_data_only
            + sys.dbms_logmnr.no_rowid_in_stmt);

    dbms_output.PUT_LINE('LogMiner started: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));
end startLogMiner;

procedure queryLogMiner(t1 varchar2, t2 varchar2, user varchar2,
        stmtfound out number) is
      outfile utl_file.file_type;
begin
```

```
    dbms_output.PUT_LINE('Querying LogMiner View: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

--        outfile := utl_file.fopen('/tmp ','out.sql','W',256);
    stmtfound := 0;
    for statement in ( select SQL_REDO, TIMESTAMP, RS_ID, SSN
                         from V$LOGMNR_CONTENTS
                          where TIMESTAMP > t1
                           AND  TIMESTAMP <= t2
                           AND SEG_OWNER = user
                           ORDER BY TIMESTAMP ASC
                     )
    loop
        dbms_output.put_line(statement.TIMESTAMP || ':'
                                || statement.RS_ID || ':'
                                || statement.SSN || ': '
                                || statement.SQL_REDO);
    stmtfound := stmtfound + 1;
--            utl_file.put_line(outfile,statement.SQL_REDO,true);
    end loop;

--        utl_file.fclose(outfile);
    dbms_output.PUT_LINE('Query finished: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

end queryLogMiner;

procedure closeLogMiner is
begin
    dbms_output.PUT_LINE('Closing LogMiner: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));

    sys.dbms_logmnr.end_logmnr();

    dbms_output.PUT_LINE('LogMiner closed: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));
end closeLogMiner;
```

```
procedure insertIntoTestTable is
begin
    insert into CONSTABLE values ('Gianni', 32);
    dbms_output.PUT_LINE('Row inserted: ' ||
                  to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS'));
    commit;
end insertIntoTestTable;

procedure test1(elapsedTime out number, stmtfound out number,
                user varchar2) is

    t1 timestamp;
    t2 timestamp;
    tStart timestamp;
    tEnd timestamp;
    secFromMinutes number;
    seconds number;
    result varchar2(6);
    interval INTERVAL DAY(0) TO SECOND(3);
begin
    t1 := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    sys.dbms_lock.sleep(3);
    insertIntoTestTable;
    sys.dbms_lock.sleep(3);
    t2 := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    startLogMinerCMine(to_char(t1,'DD-MON-YYYY HH24:MI:SS'),
                  to_char(t2,'DD-MON-YYYY HH24:MI:SS'));
    queryLogMiner(to_char(t1,'DD-MON-YYYY HH24:MI:SS'),
                  to_char(t2,'DD-MON-YYYY HH24:MI:SS'),
                  user, stmtfound);
    tStart := t2;
    tEnd := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    interval := (tEnd - tStart);
    seconds := extract(second from interval);
    secFromMinutes := 60 * extract(minute from interval);
    elapsedTime := seconds + secFromMinutes;
```

```
    if stmtfound != 1 then
        result := 'ERROR';
    else
        result := 'OK';
    end if;
    dbms_output.PUT_LINE(result || ', Elapsed time: '
                    || to_char(interval,'hh:mm:ss') || ' sec');
    closeLogMiner;
end test1;

procedure test2(elapsedTime out number, stmtfound out number,
                user varchar2) is

    t1 timestamp;
    t2 timestamp;
    tStart timestamp;
    tEnd timestamp;
    secFromMinutes number;
    seconds number;
    result varchar2(6);
    interval INTERVAL DAY(0) TO SECOND(3);
begin
    t1 := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    sys.dbms_lock.sleep(3);
    insertIntoTestTable;
    sys.dbms_lock.sleep(3);
    t2 := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    startLogMiner();
    queryLogMiner(to_char(t1,'DD-MON-YYYY HH24:MI:SS'),
                  to_char(t2,'DD-MON-YYYY HH24:MI:SS'),
                  user, stmtfound);
    tStart := t2;
    tEnd := to_timestamp(sysdate,'DD-MON-YYYY HH24:MI:SS');
    interval := (tEnd - tStart);
    seconds := extract(second from interval);
    secFromMinutes := 60 * extract(minute from interval);
    elapsedTime := seconds + secFromMinutes;
```

```
    if stmtfound != 1 then
        result := 'ERROR';
    else
        result := 'OK';
    end if;
    dbms_output.PUT_LINE(result || ', Elapsed time: '
                    || to_char(interval,'hh:mm:ss') || ' sec');
    closeLogMiner;
end test2;

procedure test1n(user varchar2, times number) is
    avgtime number;
    aggrtime number;
    onetime number;
    counter number(2);
    stmtfound number;
    result varchar2(6);
begin
    counter := 0;
    aggrtime := 0;
    stmtfound := 1;
    while counter < times and stmtfound = 1 loop
        test1(onetime, stmtfound, user);
        aggrtime := aggrtime + onetime;
        counter := counter + 1;
        result := 'OK';
        sys.dbms_lock.sleep(3);
    end loop;
    if stmtfound != 1 then
        result := 'ERROR';
    end if;
    avgtime := round(aggrtime / times, 2);
    dbms_output.PUT_LINE(result || ':' || counter
             || ', Average query time: ' || to_char(avgtime));
end test1n;

procedure test2n(user varchar2, times number) is
```

```
    avgtime number;
    aggrtime number;
    onetime number;
    counter number(2);
    stmtfound number;
    result varchar2(6);
begin
    counter := 0;
    aggrtime := 0;
    stmtfound := 1;
    while counter < times and stmtfound = 1 loop
        test2(onetime, stmtfound, user);
        aggrtime := aggrtime + onetime;
        counter := counter + 1;
        result := 'OK';
        sys.dbms_lock.sleep(3);
    end loop;
    if stmtfound != 1 then
        result := 'ERROR';
    end if;
    avgtime := round(aggrtime / times, 2);
    dbms_output.PUT_LINE(result || ':' || counter
              || ', Average query time: ' || to_char(avgtime));
end test2n;

--one time only procedure
begin

   execute immediate
     'ALTER SESSION SET nls_date_format = "DD-MON-YYYY HH24:MI:SS"';


end LogMinerUtils_pkg;
/
```

Bibliography

[1] Cgsi repository.
http://castor.web.cern.ch/castor/cgi-bin/cvsweb/cvsweb.cgi/CGSI_GSOAP/.

[2] The constanza project web page.
http://pucciani.web.cern.ch/pucciani/constanza/.

[3] Cool doxygen documentation.
http://lcgapp.cern.ch/doxygen/.

[4] The cool project.
http://lcgapp.cern.ch/project/CondDB/.

[5] Coral, common relational abstraction layer.
http://pool.cern.ch/coral/.

[6] Csim.
http://www.csim.com/.

[7] The egee project, enabling grid for e-science.
http://www.eu-egee.org/.

[8] Egee user and application portal.
http://egeena4.lal.in2p3.fr/.

[9] Emulab - network emulation testbed.
http://www.emulab.net/.

[10] Enhydra octpus: Jdbc data transormations.
http://www.enhydra.org/tech/octopus/.

[11] European datagrid project.
http://eu-datagrid.web.cern.ch/eu-datagrid.

[12] glite file transfer service.
http://www.gridpp.ac.uk/wiki/EGEE_File_Transfer_Service.

[13] Glite, lighweight middleware for grid computing.
http://glite.web.cern.ch/glite/.

[14] The globus alliance. http://www.globus.org.

[15] The globus toolkit.
http://www.globus.org/toolkit.

[16] The gram project.
http://dev.globus.org/wiki/GRAM.

[17] The grid security infrastructure.
http://www.globus.org/security/.

[18] Gt 4.0 drs: User guide.
http://www.globus.org/toolkit/docs/4.0/techpreview/datarep/.

[19] Gt 4.0 gridftp.
http://www.globus.org/toolkit/docs/4.0/data/gridftp/.

[20] Gt information services: Monitoring & discovery system (mds).
http://www.globus.org/toolkit/mds/.

[21] Ibm db2 replication.
http://www-306.ibm.com/software/data/db2/.

[22] Lcg-3d service setup, policy proposals, best practices.
https://twiki.cern.ch/twiki/bin/view/PSSGroup/ServiceDocs.

[23] The lex & yacc page.
http://dinosaur.compilertools.net/.

[24] The magic application.
http://wwwmagic.mppmu.mpg.de/introduction/.

[25] Models of networked analysis at regional centres for lhc experiments.
http://monarc.web.cern.ch/MONARC/.

[26] Myproxy, credential management service.
http://grid.ncsa.uiuc.edu/myproxy/.

[27] Nordugrid, grid solutions for wide area computing and data handiling.
http://www.nordugrid.org/.

[28] Octopus for atlas.
http://hrivnac.web.cern.ch/hrivnac/Activities/Packages/Octopus/.

[29] Ogsa-dai.
http://www.ogsadai.org.uk/index.php.

[30] Open grid forum.
http://www.ggf.org/.

[31] Open science grid.
http://www.opensciencegrid.org/.

[32] Openldap.
http://www.openldap.org/.

[33] Oracle real application cluster.
www.oracle.com/technology/products/database/clustering/index.html.

[34] Oracle streams - feature overview.
http://download.oracle.com/docs/.

[35] Overview of materialized views.
http://download.oracle.com/docs/.

[36] The planck@egee project web site.
wwwas.oat.ts.astro.it/planck-egee.

[37] Relational grid monitoring architecture.
http://www.r-gma.org/index.html.

[38] Replication (sql server 2000): Snapshot replication.
http://msdn2.microsoft.com/en-us/library/aa256286.aspx.

[39] Rfc-2459, internet x.509 public key infrastructure certificate and crl profile.
http://www.ietf.org/rfc/rfc2459.txt.

[40] Using logminer to analyze redo log files.
http://download.oracle.com/docs/.

[41] Worldwide lhc computing grid.
`http://lcg.web.cern.ch/LCG/`.

[42] Workshop on grid data replication, consistency and requirements.
`http://www.pi.infn.it/constanza/workshop/`, 26 May 2006.

[43] Domenici A., Donno F., Pucciani G., and Stockinger H. Relaxed data consistency with constanza. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid06), Singapore*, 2006.

[44] Domenici A., Donno F., Pucciani G., Stockinger H., and Stockinger K. Replica consistency in a data grid. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, Volume 534(Issues 1-22), 2004.

[45] Bakker B. Log for c++.
`http://log4cpp.sourceforge.net/`.

[46] Nielsen H.F. Barton J.J., Thatte S. Soap messages with attachments, w3c note 11 december 2000.
`http://www.w3.org/TR/SOAP-attachments`.

[47] Goodman N. Bernstein P.A. The failure and recovery problem for replicated databases. In *In Proc of the 2nd Symp. on Principles of Distributed Computing (PODC),*, 1983.

[48] Sciolla C. Implementazione e valutazione di un sistema di trasferimento file basato su soap in ambiente grid. Master's thesis, Universitá di Pisa, Facoltá di Ingegneria, 2007.

[49] Dantressangle P. Chen Y., Berry D. Transaction-based grid database replication. In *In Proc. of the UK e-Science All Hands Meeting 2007*, 2007.

[50] The CMS Collaboration. Cms computing, software and analysis challenge in 2006 (csa06) summary, 7 march 2007.
`cms.cern.ch/iCMS/`.

[51] Skeen D. Nonblocking commit protocols. In *In Proc. Of ACM SIGMOD Int. Conf. on Management of Data*, 1981.

[52] Codd E.F. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377 – 387, 1970.

[53] Bell W. et al. Optorsim - a grid simulator for studying dynamic data replication strategies. *Int. Journal of High Performance Computing Applications*, 17(5), 2003.

[54] Burke S. et al. Glite 3 user guide. `https://edms.cern.ch/file/722398/1.1/gLite-3-UserGuide.pdf`, 2007.

[55] Cederqvist P. et al. Version management with cvs. `http://ximbiot.com/cvs/manual/`.

[56] Duellmann D. et al. Models for replica synchronisation and consistency in a data grid. In *In Proc. of the 10th IEEE Symposium on High Performance and Distributed Computing (HPDC)*, 2001.

[57] Duellmann D. et al. Lcg 3d project status and production plans. In *In Proc. of International Conference on Computing in High Energy and Nuclear Physics, TIFR, Mumbai, India*, 2006.

[58] Gervasi O. et al. A grid molecular simulator for e-science, book chapter, lncs volume 3470. *Lecture Notes In Computer Science*, 3470:16 – 22, 2005.

[59] Gray J. et al. The dangers of replication and a solution. In *In Proc of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173 – 182, 1996.

[60] Holliday J. et al. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218 – 1238, 2003.

[61] Kosyakov S. et al. Frontier: High performance database access using standard web components in a scalable multi-tier architecture. In *In Proc. of International Conference on Computing in High Energy and Nuclear Physics, (CHEP '04), Interlaken, Switzerland, 27 Sep - 1 Oct 2004*, 2004.

[62] Nedim Alpdemir M. et al. Ogsa-dqp: A service-based distributed query processor for the grid. In *In Proc. of UK e-Science All Hands Meeting Nottingham. EPSRC*, 2003.

[63] Terry D.B. et al. Session guarantees for weakly consistent replicated data. In *In Proc of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30*, pages 140 – 149. IEEE Computer Society, 1994.

[64] Mattern F. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 120 – 131, 1989.

[65] Pacini F. Job description language how to. `http://www.infn.it/workload-grid/docs/`.

[66] Casanova H. Distributed computing research issues in grid computing. *ACM SIGAct News*, 33(3):50 – 70, 2002.

[67] Stockinger H. Defining the grid: A snapshot on the current view. *Journal of Supercomputing*, 42(1):3 – 17, 2007.

[68] Yu H. and Vahdat A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239 – 282, 2002.

[69] Foster I. and Kesselman C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, 1998.

[70] Foster I., Kesselman C., Nick J., and Tuecke S. The physiology of the grid: An open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[71] Papadopulos I. Pool: the lcg persistency framework. IEEE Nuclear Science Symposium, Portland, Oregon, 2003.

[72] Gray J. and Reuter A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 1993.

[73] Haas L. and Lin E. Ibm federated database technology. `http://www.ibm.com/developerworks/db2/`.

[74] Ciriello M. Integrazione della grid security infrastructure in un servizio di consistenza. Master's thesis, Universitá di Pisa, Facoltá di Ingegneria, 2006.

[75] Ozsu M.T. and Valduriez P. *Principles of distributed database systems*. Prentice-Hall, Inc., 1991.

[76] Cottingham N.W. and Greenwood A.D. *An Introduction to the Standard Model of Particle Physics*. Cambridge University Press, 1999.

[77] Alsberg P. and Day J. A principle for resilient sharing of distributed resrources. In *In Proc. Of the 2nd Int. Conf. on Software Engg., San Francisco, CA., pp. 562 - 570*, 1976.

[78] Bernstein P.A. and Goodman N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9:596 – 615, 1984.

[79] Bernstein P.A., Hadzilacos V., and Goodman N. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[80] Brun R. and Rademakers F. Root: An object oriented analysis framework. In *In Proc. of the 5th International Workshop On New Computing Techniques In Physics Research, EPFL Lausanne (Switzerland)*, 1996.

[81] Jih-Sheng Chang R-Shiung Chang. Adaptable replica consistency service for data grids. In *In Proc. of the 3rd International Conference on Information Technology: New Generations (ITNG'06)*, 2006.

[82] Jain R.K. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[83] Susarla S. and Carter J. Middleware support for locality-aware wide area replication. Technical Report UUCS-04-017, School of Computing, University of Utah.

[84] Susarla S. and Carter J. Flexible consistency for wide-area peer replication. In *In Proc. of the 25th Annual International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[85] Goel S. Taniar D. Concurrency control issues in grid databases. *Future Generation Computer Systems*, 23(1), 2007.

[86] Gallivan K. Van Engelen R.A. The gsoap toolkit for web services and peer-to-peer computing networks. In *In Proc. of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 128 – 135, 2002.

[87] ATLAS working group. Atlas computing, technical design report. `http://doc.cern.ch/archive/electronic/cern/`, 2005.

[88] Breitbart Y. and Korth H.F. Replication and consistency: being lazy helps sometimes. In *In Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173 – 184, 1997.

[89] Saito Y. and Shapiro M. Optimistic replication. *ACM Computing Surveys*, 37(1):42 – 81, 2005.

[90] Zhiwei Xu Yuzhong Sun. Grid replication coherence protocol. In *In Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 13*, 2004.